

3

GPIO e Assembly

Versão 2.0

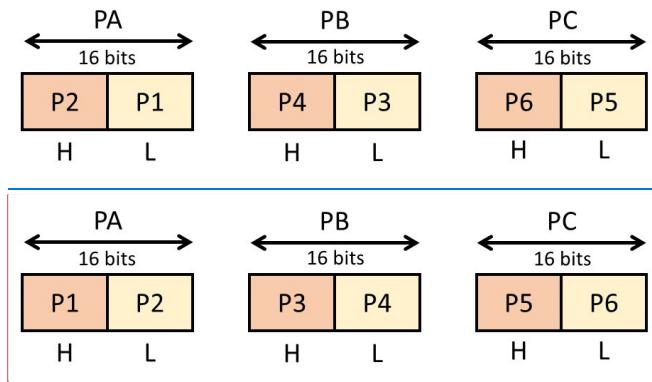
O termo GPIO, do inglês *General Purpose Input/Output*, pode ser traduzido como Entrada/Saída de Finalidade Geral. Ele se refere aos pinos do processador que podem ser usados com entrada ou saída digital. O programador pode configurar como saída qualquer um desses pinos e gerar uma saída em 1 (nível alto) ou em 0 (nível baixo), quando quiser. Por outro lado, pode-se configurar como entrada qualquer um desses pinos e lê-lo para saber se ele está recebendo um 1 (nível alto) ou um 0 (nível baixo). Em suma, o GPIO é usado para criar entradas e saídas digitais.

3.0. Quero Usar o GPIO do MSP430 e Não Pretendo Ler Todo Este Capítulo

Sob este título, é apresentado um resumo do GPIO do MPS430. Para evitar confusão como o restante do capítulo, as figuras aqui apresentadas não são numeradas e, algumas vezes, são uma mera repetição de outras figuras.

A família MSP430 pode ter até 12 portas, numeradas de P1 até P11 e ainda a porta PJ. A identificação dos bits de cada porta segue representação **Porta.Bit**. Assim, P7.5 significa bit 5 da porta P7. As portas também podem ser acessadas em 16 bits. Neste caso, elas são agrupadas duas a duas, como mostrado na figura abaixo e rotuladas com letras. A lógica de denominação e numeração dos bits continua a mesma, por exemplo, PA.11 corresponde ao bit 11 da porta PA, que é o P1P2.3. Vamos trabalhar apenas com ♫

acesso de 8 bits e usaremos a designação genérica **Pn.x**, onde **n** indica o número (ou a letra) da porta e o **x**, o bit.



Comentado [dcc1]: A significância está ao contrário. Dentro de PA, P1 é o LSByte e P2 é o MSByte (vide tabela 12-2 do User's Guide)

A operação das portas é feita com os registradores listados abaixo, um para cada porta. No final deste capítulo está um gabarito com todos esses registradores para facilitar a configuração e o uso dessas portas.

Registrador	Função	PnDIR = 0 (entrada)	PnDIR = 1 (saída)
PnDIR	Direção	= 0 → entrada	= 1 → saída
PnIN	Leitura <u>de valores da entrada</u>	Ler estado do pino	<u>Ler estado do pino</u> Teste de loopback: permite verificar se o que escrevemos em OUT foi de fato escrito.
PnOUT	<u>Estado da saída</u> Escrita de <u>valores na saída</u>	Pull up/down	<u>Estado do pino</u> Valor lógico do pino na saída
PnDS	Habilitar driversConfigura capacidade de fornecer corrente	-	Driver low/high = 0 → ~5mA = 1 → ~15mA
PnREN	Habilitar resistor	Habilita resistor	-
PnSEL	Função alternativa	= 0 → GPIO = 1 → função alternativa	= 0 → GPIO = 1 → função alternativa

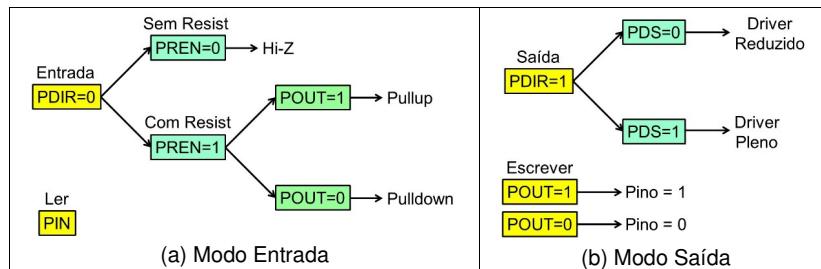
n = 0, 1, 2, ...

Formatado: Fonte: Negrito

Além de trabalhar como GPIO, a grande maioria dos pinos das diversas portas pode ser usada para uma outra finalidade, que aqui chamaremos de função alternativa. Isto fica claro nas Figuras 3.1 e 3.15. Para disponibilizar essa função alternativa pelo pino, o bit correspondente no registrador PnSEL deve estar em 1.

Formatado: Fonte: Negrito

Ao usarmos as portas como GPIO, precisamos definir quais serão entradas e quais serão saídas e seus valores lógicos. As ilustrações a seguir resumem as opções de operação.



A seguir estão 4 exemplos de códigos para as operações mais comuns com o bit de uma porta. Usamos P1.2 como exemplo.

Fazer P1.2 = 0	Fazer P1.2 = 1
<pre>BIS.B #BIT2,&P1DIR ;P1.2 = saída BIC.B #BIT2,&P1OUT ;P1.2 = 0</pre>	<pre>BIS.B #BIT2,&P1DIR ;P1.2 = saída BIS.B #BIT2,&P1OUT ;P1.2 = 1</pre>
Fazer P1.2 = entrada c/ pullup e testar P1.2	Fazer P1.2 = entrada Hi-Z e testar P1.2
<pre>BIC.B #BIT2,&P1DIR ;P1.2 = entrada BIS.B #BIT2,&P1REN ;Hab resistor BIS.B #BIT2,&P1OUT ;Selec. pullup BIT.B #BIT2,&P1IN ;Testar P1.2</pre>	<pre>BIC.B #BIT2,&P1DIR ;P1.2 = entrada BIC.B #BIT2,&P1REN ;Desab. resistor BIT.B #BIT2,&P1IN ;Testar P1.2</pre>

Tabela formatada

Formatado: À esquerda

Formatado: À esquerda

Formatado: À esquerda

Formatado: À esquerda

Após a leitura deste pequeno texto, recomenda-se o estudo dos exemplos apresentados no final do capítulo. Em todo o caso, é fortemente recomendada a leitura de todo este capítulo.

3.1. Introdução

A Figura 3.1 apresenta a pinagem da CPU MSP430F5529. São 80 pinos com as mais diversas finalidades. Nessa figura procure, por exemplo, pelos pinos marcados por P1.0, P1.1, ..., P1.7 (são os pinos numerados de 21 até 28). Esses são os 8 pinos que dão acesso à porta P1. Para identificar as portas e seus elementos, usaremos a representação de **Porta.Bit**. Assim, P1.0 significa bit 0 da porta P1, P1.1 significa bit 1 da porta P1, e assim por diante, como mostrado na Figura 3.2. Isso vale para qualquer porta, por exemplo, P7.5 (pino 58) significa bit 5 da porta P7.

Na Figura 3.1, o leitor deve ter notado que cada pino da CPU, além do GPIO, é usado também para uma função especial. Por exemplo, o pino 22, além de P1.1, também pode ser usado para o Comparador 0 do Timer A0 (TA0.0). Tudo isso será estudado mais adiante. Por hora, vamos nos dedicar apenas às portas operando como I/O.

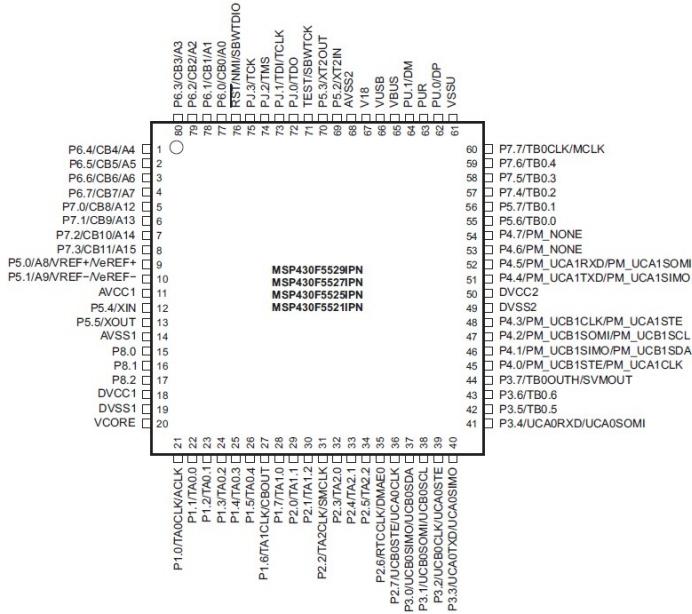


Figura 3.1. Pinagem da CPU MSP430F5529.

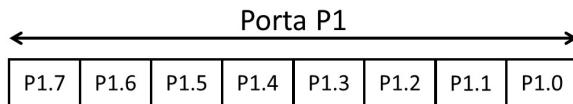
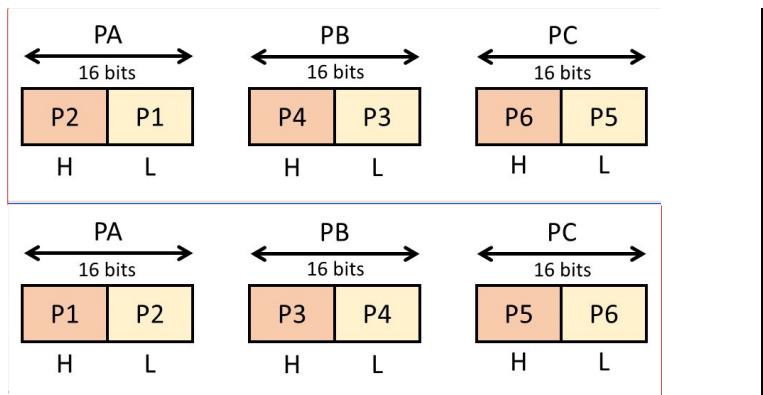


Figura 3.2. Numeração dos 8 bits da porta P1.



Comentado [dcc2]: Corrigir significância das portas

Figura 3.3. Acesso ao GPIO em 16 bits.

Os acessos via P1, P2 etc. são feitos em 8 bits. As portas também podem ser acessadas em 16 bits. Neste caso, elas são agrupadas duas a duas, como mostrado na Figura 3.3. A lógica de denominação e numeração dos bits continua a mesma, por exemplo, PA.11 corresponde ao bit 11 da porta PA, que é o P1P2.3. Um erro muito comum é pensar que o pino P1P2.0 corresponde ao PA.0, quando, na verdade, ele é o P1PA.8.

A LaunchPad que usamos não disponibiliza os bits de todas as portas. A Tabela abaixo indica os que estão disponíveis.

Tabela 3.1. Os pinos marcados com "*" estão disponíveis na LaunchPad

	7	6	5	4	3	2	1	0	Total
P0	-	-	-	-	-	-	-	-	0
P1	-	*	*	*	*	*	-	-	5
P2	*	*	*	*	*	*	-	*	6
P3	*	*	*	*	-	*	*	*	6
P4	-	-	-	-	*	*	*	*	4
P5	-	-	-	-	-	-	-	-	0

Tabela formatada

P6	-	*	*	*	*	*	*	*	6
P7	-	-	-	*	-	-	-	*	2
P8	-	-	-	-	-	-	*	-	1

← Tabela formatada

A família MSP430 pode ter até 12 portas, numeradas de P1 até P11 e ainda a porta PJ (J-Tag). Cada bit de cada porta pode ser configurado individualmente como entrada ou como saída. Antes de passarmos ao estudo dessas portas vamos abordar dois conceitos que são fundamentais para sua compreensão.

3.2. Pullup/Pulldown e Schmitt Trigger

Iniciamos de forma genérica. A entrada de um circuito digital CMOS, como é o caso do MSP430, tem alta impedância de entrada. Isto significa que quando solta (não conectada) fica muito suscetível a ruído o que provoca um comportamento errático e aumenta o consumo de energia. Usualmente não devemos deixar tais entradas “soltas”. A solução mais simples é o uso de um resistor para definir o valor desta entrada, conectando-a ao Vcc ou ao terra, como mostrado na Figura 3.4.

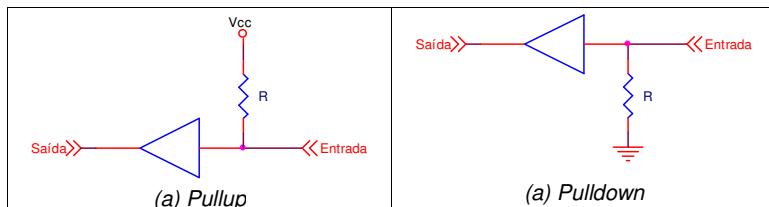


Figura 3.4. Duas soluções para não deixar uma entrada CMOS desconectada.

Os termos são óbvios. Usa-se *pullup* (do inglês puxar para cima) para indicar que se usou um resistor para puxar a entrada para nível alto (Vcc) e o termo *pulldown* (do inglês puxar para baixo) para indicar que se usou um resistor para puxar a entrada para nível baixo (terra).

Como dito anteriormente, o resistor de pull-up/down força um valor a um pino que está flutuando, ou seja, não está conectado a nada. Isso é muito comum de acontecer quando usamos botões do tipo push-button. Quando pressionamos o botão, o pino faz contato, porém quando o botão está solto, o pino não faz contato com nada. Dizemos então que o pino do microcontrolador fica flutuando.

Vamos avaliar a corrente que passa no resistor nesse caso: Quando o pino de entrada não está conectado a nada, a corrente do lado da "entrada" é zero. Do lado do buffer, a corrente também é zero, pois tipicamente se trata do gate de transistores MOS com elevada impedância. Como essas duas correntes são zero, a corrente que passa pelo resistor também é zero e isso faz com que a queda de tensão do resistor seja zero. Dessa forma, o resistor reflete a tensão da sua extremidade no pino de entrada, VCC, ou 1 lógico para o resistor de pull-up e GND, ou 0 lógico para o resistor de pull-down.

Podemos concluir então que o resistor de pull-up/down permite atribuir um valor padrão a um pino quando ele não está conectado a nada. Assim que o valor da entrada mudar de desconectado para qualquer valor, seja ele 0, ou 1 lógico, o valor padrão estabelecido pelo resistor é sobreescrito sem gerar qualquer confusão, curto ou ruído.

Resistores de pull-up/down não são usados para remover ruído de chaveamento. Estudaremos esse problema específico mais à frente.

O uso de *pullup* ou *pulldown* na entrada de uma porta, não impede seu uso, como mostrado na Figura 3.5. Isso porque são usados resistores de valor elevado (acima de 10 kΩ). É claro que se forem usados resistores de baixo valor, a entrada poderá ficar "presa" num determinado nível lógico.

Comentado [dcc3]: Não sei se vale a pena comentar isso sem fazer uma análise de circuito. Acho que confunde mais do que ajuda. Recomendo remover este parágrafo.

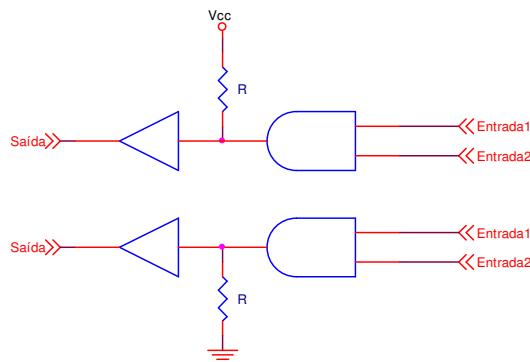


Figura 3.5. É possível usar as entradas cujos níveis foram definidos com resistores de *pullup* ou *pulldown*.

Passemos ao segundo conceito que é importante para se entender o funcionamento do GPIO do MSP430. Os níveis digitais com os quais as portas do MSP430 F5529 operam, quando o microcontrolador está alimentado com 3,3 V estão mostrados na Figura 3.6.

Explicando melhor. Quando uma tensão é aplicada a uma entrada do MSP, ela será interpretada como o lógico 0 se estiver abaixo de 0,9 V ou como o lógico 1 se estiver acima de 1,9 V. A faixa entre 0,9 V e 1,9 V é chamada de faixa proibida (ou indeterminada) e define a margem de ruído na entrada. Ruídos capazes de induzir tensões acima de 1,0 V podem fazer um 0 “virar” 1 ou ao contrário. No caso da saída, o fabricante garante que quando uma saída do MSP estiver em lógico 0, ela estará com uma tensão abaixo de 0,25 V e quando a saída for o lógico 1, ela está com uma tensão acima de 3,05 V. A margem de ruído na saída é de 2,8 V.

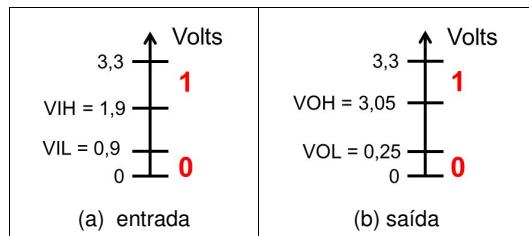


Figura 3.6. Definição dos níveis lógicos para as entradas e para as saídas digitais do MSP430 F5529 quando alimentado com 3,3 V (gráficos fora de escala).

Para facilitar a explicação, vamos imaginar uma porta cuja entrada não tenha a faixa proibida. Uma porta onde a entrada é interpretada como o lógico 0 quando ela for menor ou igual a 1,5 V ou como o lógico 1 quando estiver acima da 1,5 V. O que será que acontece com essa porta se a ela aplicarmos um sinal como mostrado na Figura 3.7? É simples: enquanto a tensão do sinal estiver abaixo de 1,5 V, ele é interpretado como 0 e a saída, então vai para 0. No instante “t” a entrada ultrapassa 1,5 V e passa a ser interpretada como 1 e por isso a saída vai para 1.

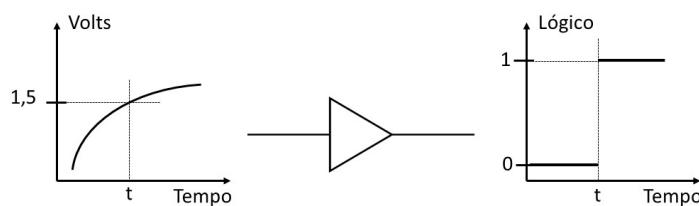


Figura 3.7. Aplicação de um sinal analógico à entrada de uma porta digital hipotética cujo limiar de transição na entrada igual a 1,5 V.

Porém, precisamos nos lembrar do ruído que sempre está presente. Qual será o resultado na saída se a entrada estiver corrompida com “um pouco” de ruído. A Figura 3.8 tenta ilustrar esta situação. Quando o sinal de entrada estiver se aproximando de 1,5 V, o ruído sempre presente provocará oscilações acima e abaixo de deste valor e o resultado na saída serão rápidas transições entre os lógicos 0 e 1, antes de se estabilizar. Essas transições podem “enganar” uma decisão baseada no nível lógico da saída desta porta.

Quando se trabalha com uma porta real, o problema ainda é pior porque ela possui a faixa proibida e o comportamento da porta quando a tensão do sinal de entrada está dentro desta faixa não é garantido. Precisamos então de uma porta que apresente um comportamento melhor quando a tensão de entrada está próxima da tensão de transição ou da faixa proibida.

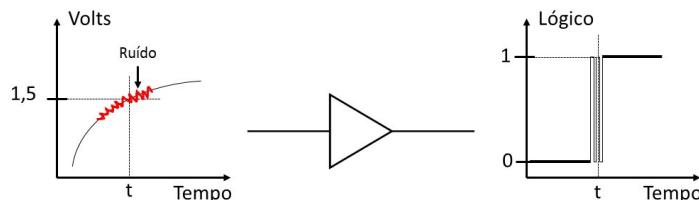


Figura 3.8. Aplicação de um sinal ruidoso à entrada de uma porta digital hipotética cujo limiar de transição na entrada igual a 1,5 V.

Existe uma classe de portas lógicas, denominada Schmitt-trigger que foram projetadas para que o nível lógico de sua entrada fosse interpretado de forma interessante. Vamos usar como exemplo uma porta comercial da família TTL, que é o 74LS14. Este é um inversor Schmitt-trigger cujo comportamento está ilustrado na Figura 3.9. O símbolo da porta está apresentado do lado esquerdo da figura e note neste símbolo um pequeno desenho que lembra uma curva de histerese. Esta é a ideia: a porta apresenta uma histerese em sua entrada. “O caminho de ida é diferente do caminho de volta”.

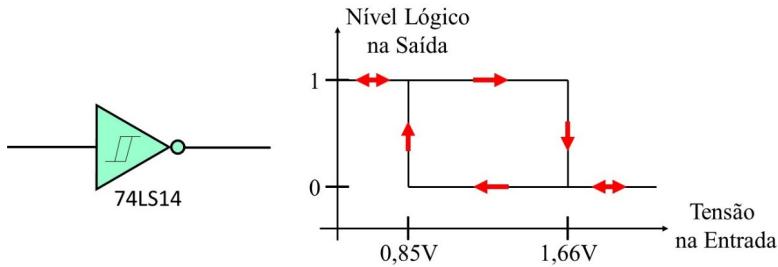


Figura 3.9. Figura apresentando o inversor Schmitt-trigger e seu gráfico de saída lógica versus tensão analógica na entrada.

Vamos começar a analisar a curva de resposta apresentada na Figura 3.9. Iniciamos com 0 V na entrada da porta. Note que a porta é uma inversora, assim, com esta entrada, a saída é o lógico 1. Vamos aumentando a tensão até o limite de 1,66 V. Enquanto estivermos abaixo de 1,66 V, a saída permanece em lógico 1. Porém, quando ultrapassarmos essa tensão a saída muda para o lógico 0. Se continuarmos a aumentar a tensão na entrada, nada se altera. Vamos agora fazer o caminho inverso e começar a diminuir a tensão na entrada. Note que quando caímos abaixo de 1,66 V nada muda. A saída só volta para o lógico 1 quando a tensão na entrada cair abaixo de 0,85 V. Essa porta vai oferecer em sua saída um sinal limpo. É claro, desde que o ruído não induza tensões acima de 0,81 V.

A Figura 3.10 ilustra esta ideia. O sinal ruidoso aplicado a um *buffer* Schmitt-trigger não inversor, como o usado no MSP, não provocou transições indesejadas por ocasião da transição do lógico 0 para o lógico 1.

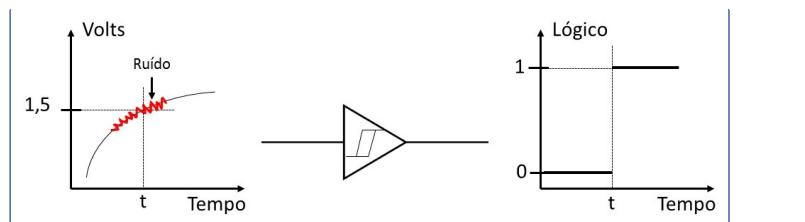


Figura 3.10. Aplicação de um sinal ruidoso à entrada de uma porta Schmitt-trigger.

Comentado [dcc4]: Adaptar a figura para incluir os limiares 0,85 e 1,66. O valor lógico só vai para 1 quando passa de 1,66.

Comentado [RZ5R4]: Será que não vai colocar muitos detalhes desnecessários na figura. A ideia é apenas fazer um contraste com a fig 3.8.

3.3. Esquema da Portas GPIO do MSP430

Vamos agora abordar a configuração e o funcionamento das portas GPIO disponíveis no MSP. Para facilitar as explicações, tomaremos como referência o bit x da porta P1, ou seja, P1.x. A Figura 3.11 apresenta um esquema muito simplificado desta porta. Por enquanto, temos 3 registradores envolvidos (na verdade, o bit x desses 3 registradores):

- P1DIR → que controla a direção da porta, 0 = entrada e 1 = saída;
- P1OUT → nível lógico que surge no pino quando a porta está no modo saída e
- P1IN → permite a leitura do pino.

Nesta figura Na figura 3.11, se P1DIR.x = 1, o *tri-state* U1 está ativado e com isso a porta está no modo saída. Neste caso, o estado de P1OUT.x passa pelo buffer não inversor U1 e surge num determinadono pino P1.x do MSP.

O estado deste pino do pino P1.x pode ser lido com o uso de P1IN.x, que recebe seu estado após passar por um *buffer Schmitt-trigger* não inversor U2. Neste caso No caso de um pino configurado como saída, a leitura de P1IN.x vai resultar no estado de P1OUT.x, ou seja, o programador vai ler de volta o que ele programou em P1OUT.x. Esse tipo de ação é conhecido como teste de loopback, muito comum para testar se o pino está de fato funcionando ou não.

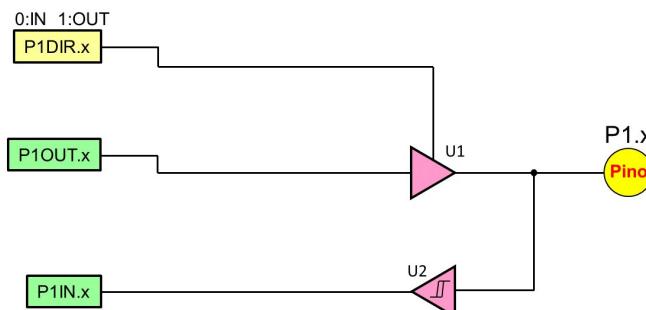


Figura 3.11. Esquema simplificado de uma porta do MSP.
No caso foi tomado como exemplo o bit x da porta P1 (P1.x).

A Figura 3.12 apresenta um detalhe sobre o modo saída. Quando uma saída é configurada, o programador pode escolher entre usar um *driver* com baixa capacidade de corrente (U1LD) e baixo consumo ou um *driver* com alta capacidade de corrente (U1HD) e alto consumo. Isto é feito com o registrador P1DS. Note que, graças às duas portas AND, a seleção só acontece quando P1DIR.x = 1, que é o modo saída. Essa opção de seleção da capacidade do *driver* atende ao perfil de baixo consumo deste processador. O usuário ativa o *driver* de alto consumo somente quando necessitar.

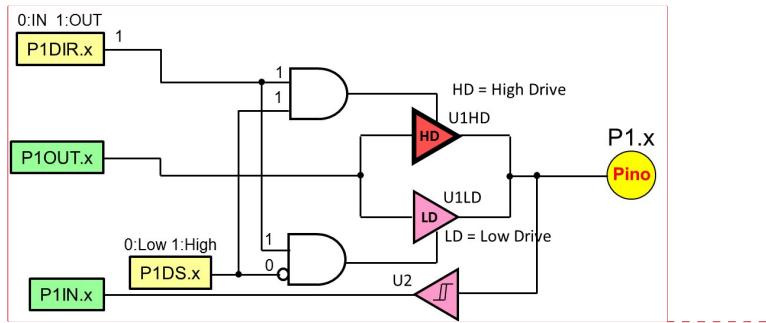


Figura 3.12. Esquema simplificado de uma porta do MSP, ressaltando os drivers de alto (HD) e baixo (LD) desempenho.

Comentado [dcc6]: Do ponto de vista didático, acho que simplificaria bastante se você colocar o pino DS como um switch extra ao buffer de saída, como se fosse um buffer de ganho programável.

Apenas um comentário. Acho que está bom dessa forma.

Para configurar um pino como entrada, é preciso fazer $P1DIR.x = 0$. Quando isto acontece, o driver U1 é desabilitado, como mostrado na Figura 3.13. Assim, a entrada fica em alta impedância (*Hi-Z*) e pode ser lida em $P1IN.x$. Isto é útil em algumas-situações que o pino é conectado à um dispositivo que nunca deixa o pino flutuando, ou seja, nunca fica desconectado. Em situações , mas em geral, como estudamos no item anterior, em que o pino fica flutuando, como, por exemplo, quando está conectado à um push-button, ou num barramento com mais de 2 dispositivos comunicantes. -é interessante forçar este pino para nível alto ou para nível baixo. O MSP oferece recursos neste sentido, como mostrado na Figura 3.14.

Formatado: Fonte: Itálico

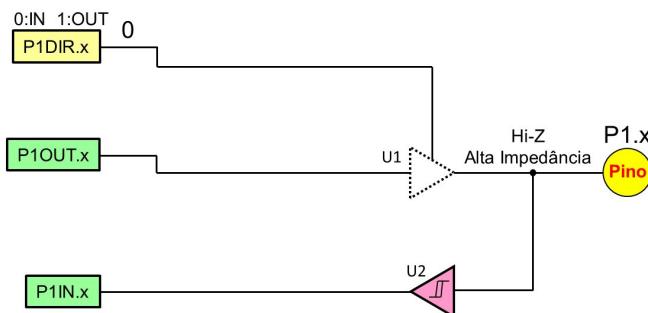


Figura 3.13. Esquema simplificado para quando o bit x da porta P1 ($P1.x$) é configurado como entrada.

Toda a configuração de quando o bit x da porta P1 é usado como entrada ($P1DIR.x = 0$), está esquematizado no Figura 3.14. O programador pode desejar usar o pino configurado para alta impedância, neste caso ele faz $P1REN.x = 0$, com isso a chave SW fica aberta,

pois a saída da porta U3 vai para 0. Por outro lado, para definir um nível de tensão para um pino configurado como entrada, é preciso fazer $P1REN.x = 1$. Neste caso, a chave SW fecha, pois, a saída de U3 vai para 1. Agora, com o uso do multiplexador analógico U4, o programador define se quer *pullup* ($P1OUT.x = 1$) ou *pulldown* ($P1OUT.x = 0$). É interessante notar que, quando configurado como entrada, o *bit* que era usado para definir o estado da saída ($P1OUT.x$) ficou ocioso. Então, ele é aproveitado para definir *pullup* ou *pulldown*. Note que a porta U3 garante que a chave SW fique aberta quando o pino é configurado no modo saída ($P1DIR.x = 1$).

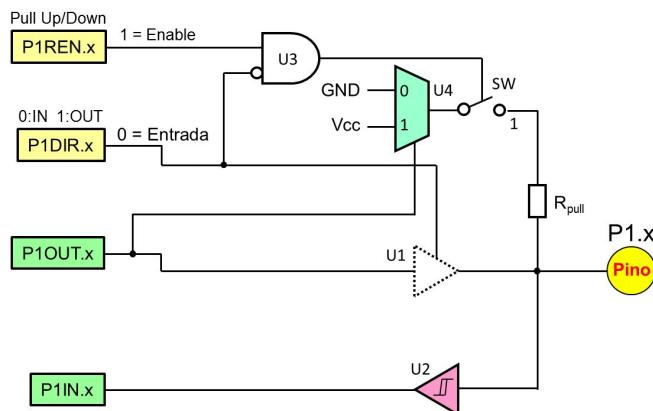


Figura 3.14. Possibilidade de uso de *pullup* ou *pulldown* pelo bit x da porta P1 ($P1.x$) quando configurada como entrada.

Com isso completamos a descrição essencial de um *bit* de GPIO. Entretanto, como pode ser visto na Figura 3.1, além de GPIO, os bits das portas são usados para diversas outras funções, que aqui vamos denominar de forma genérica como Funções Alternativas.

Para permitir que um pino fique dedicado a uma Função Alternativa, existe o registrador P1SEL, como mostrado na Figura 3.15. Se $P1SEL.x = 0$, os multiplexadores U5 e U6 garantem que todo o controle seja feito pelos registradores P1DIR.x e P1OUT.x, como foi estudado até aqui. Mas, se $P1SEL.x = 1$, esses mesmos multiplexadores, U5 e U6, entregam o comando para o MSP que pode controlar a direção e o estado da saída. Note ainda que existe o *buffer* U7 que permite ao MSP ler o estado do pino.

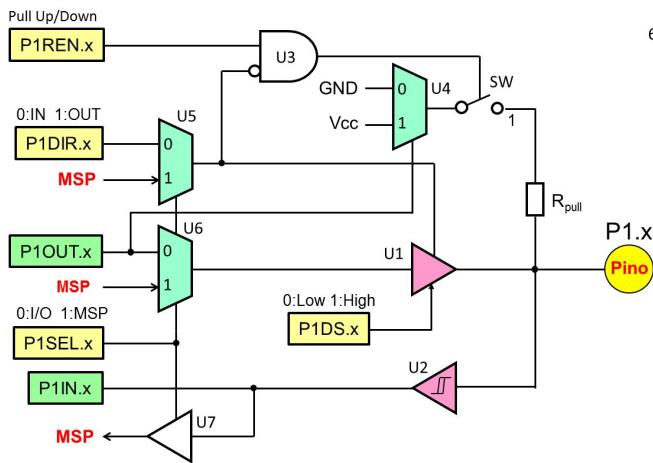


Figura 3.15. Esquema completo do bit x da porta P1 (P1.x).

Comentado [DC7]: Será que não seria melhor usar o nome “periférico dedicado” ao invés de “MSP”. Apenas uma sugestão. Acho que está bom dessa forma. A explicação do texto está bem boa e de acordo com a imagem.

A Tabela 3.2 resume todos os registradores usados para definir os modos de operação dos recursos de GPIO. A letra “n” deve ser substituída pelo número da porta. Lembramos de que cada bit da porta ($P_n.x$) pode ser configurado individualmente. A Figura 3.16 resume as opções de configuração. Note que nesta figura, para beneficiar a clareza, omitimos o número do registrador e o número do bit.

A Tabela 3.2. Resumo dos registradores usados para controlar os recursos de GPIO.

Registrador	Função	$P_nDIR = 0$ (entrada)	$P_nDIR = 1$ (saída)
P_nDIR	Direção	$= 0 \rightarrow$ entrada	$= 1 \rightarrow$ saída
P_nIN	Leitura	Ler estado do pino	Ler estado do pino
P_nOUT	Estado da saída	Pullup/down	Estado do pino
P_nDS	Habilitar drivers	-	Driver low/high
P_nREN	Habilitar resistor	Habilita resistor	-
P_nSEL	Função alternativa	$= 0 \rightarrow$ GPIO $= 1 \rightarrow$ MSP	$= 0 \rightarrow$ GPIO $= 1 \rightarrow$ MSP

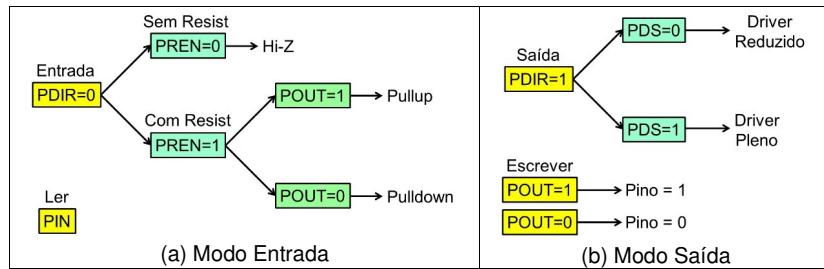
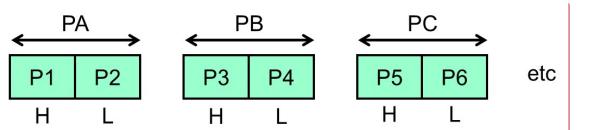


Figura 3.16. Esquema para orientar a configuração dos recursos de GPIO.

Para finalizar a parte que trata da configuração dos recursos de GPIO, vamos abordar mais alguns detalhes sobre os registradores envolvidos. Eles podem ser acessados como registradores de 8 bits, o que fizemos até agora, ou como registradores de 16, como indicado na Figura 3.17. Quando se acessa em 16 bits, os números das portas são substituídos por letras, sendo que a ideia geral é mantida. Por exemplo, continuam existindo os registradores PADIR, PAOUT, PAREN etc. Deve-se tomar cuidado com a numeração dos bits quando se mistura acessos de 8 e de 16 bits. Por exemplo, o bit P21.0 (8 bits) corresponde ao bit PA.8 (16 bits). É muito comum o erro de pensar que P21.0 corresponda ao PA.0.



Comentado [DC8]: Trocar a significância das portas

Figura 3.17. Organização dos registradores de GPIO em porções de 8 bits e 16 bits.

Tomando como exemplo os registradores de 8 bits, fica a pergunta: como fazer para acessar valores de bits individualmente ao invés de acessar o registro todo? Para isso, existem as instruções em assembly BIS e BIC de "bit set" e "bit clear" respectivamente. Essas instruções realizam operações binárias de forma modificar apenas os bits apontados por máscaras no registro de destino. Vejamos o funcionamento dessas instruções em detalhes.

Na operação booleana OU, o valor 1 lógico é predominante. Em outras palavras, 1 OU x sempre resulta em 1. Essa operação é usada para "setar" um bit, ou seja, levar o valor do bit para 1. Veja no exemplo da figura abaixo.

	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
Registro (antes)	0	1	0	1	0	0	1	1
Máscara	0	0	0	0	1	0	0	0
Registro (depois)	0	1	0	1	1	0	1	1

Aqui, queremos “setar” o BIT3 do registro. Para isso, realizamos a operação OU entre o registro e a máscara. A máscara indica os bits que queremos modificar com 1 lógico. Os valores que não forem marcados com 1, ou seja, que tiverem um 0 lógico, não serão modificados.

De maneira complementar, na operação booleana E, o valor 0 lógico é predominante. Em outras palavras 0 E x sempre resulta em 0. Podemos usar isso para zerar posições específicas de registradores como no exemplo a seguir:

	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
Registro (antes)	0	1	0	1	0	0	1	1
Máscara	1	1	1	1	1	1	0	1
Registro (depois)	0	1	0	1	0	0	0	1

Neste exemplo, queremos apenas zerar o BIT1 do registro sem alterar os demais, para isso, usamos uma máscara com um 0 apontando a posição que queremos zerar e mantemos todos os demais bits em 1.

É pouco prático usar máscaras usando 0's para apontar a posição que queremos zerar. É muito mais intuitivo usar 1's para marcar essas posições. Por sorte, a instrução BIC já inverte a máscara para nós.

Resumindo, temos as seguintes instruções para modificar bits dentro de bytes.

BIS mask, dst → dst,= dst | mask
BIC mask, dst → dst,= dst & ~mask

Se quisermos zerar o bit 5 do registro R10, usamos a instrução BIC #BIT5, R10. De maneira complementar, se quisermos setar o bit 7 do registro R10 usamos BIS #BIT7, R10.

Além dessas duas instruções, ainda temos a instrução XOR e a instrução AND que podem ser usadas para alternar o estado de um bit dentro de um byte ou filtrar um agrupamento de bits. O leitor é convidado a demonstrar esse comportamento da mesma forma que foi feito para as instruções BIS e BIC.

Apresentamos a seguir alguns casos de configuração com o pino P1.2, que foi tomado como exemplo.

Configurar P1.2 como saída e depois fazer P1.2 = 0.

```
BIS.B #BIT2,&P1DIR          ;P1DIR.2 = 1 (configura como saída)
BIC.B #BIT2,&P1OUT          ;P1OUT.2 = 0 (escreve 0 na saída)
```

Formatado: Português (Brasil)

Configurar P1.2 como saída e depois fazer P1.2 = 1.

```
BIS.B #BIT2,&P1DIR          ;P1.2 = saída
BIS.B #BIT2,&P1OUT          ;P1.2 = 1
```

Configurar P1.2 como entrada com *pullup* e ler porta P1 em R5.

```
BIC.B #BIT2,&P1DIR          ;P1.2 = entrada
BIS.B #BIT2,&P1REN          ;Habilitar resistor de P1.2
BIS.B #BIT2,&P1OUT          ;Selecionar pullup para P1.2
MOV.B &P1IN,R5              ;R5 = PORTA P1
```

Configurar P1.2 como entrada em alta impedância e ler porta P1 em R5.

```
BIC.B #BIT2,&P1DIR          ;P1.2 = entrada
BIC.B #BIT2,&P1REN          ;Desabilitar resistor de P1.2
MOV.B &P1IN,R5              ;R5 = PORTA P1
```

Dica: Apesar de termos apresentado, por questões didáticas, a manipulação do pino de saída configurando primeiro o registro DIR e depois o registro OUT, é muito comum fazer o inverso para evitar glitches na saída. Quando configuramos a direção do pino como saída, fazendo DIR = 1, não conhecemos o valor anterior do registro OUT. Se o registro de saída OUT tiver um valor diferente do desejado, um glitch de curta duração será gerado no pino de saída. Essa condição pode ser evitada configurando o registro OUT antes do registro DIR.

Formatado: Fonte: Itálico

Formatado: Fonte: Itálico

Formatado: Fonte: Itálico

3.4. Compatibilizando GPIO para Diferentes Tensões

Vamos abordar um problema muito comum: como conectar o MSP430, que é alimentado com tensão de 3,3 V, a dispositivos alimentados com 5 V? Para apresentar as ideias, vamos tomar como exemplo a conexão entre o MSP e a lógica TTL.

A Figura 3.18 apresenta a definição dos níveis lógicos usados pela família TTL, que é alimentada com 5 V e dos níveis lógicos do MSP, que é P quando alimentado com 3,3 V. Será usada a notação apresentada a seguir:

Ricardo Zelenovsky, Daniel Café e Eduardo Peixoto

- V_{OL} → máxima tensão na saída quando ela está em nível lógico 0;
- V_{OH} → mínima tensão na saída quando ela está em nível lógico 1;
- V_{IL} → máxima tensão para uma entrada ser interpretada como nível lógico 0 e
- V_{IH} → mínima tensão para uma entrada ser interpretada como nível lógico 1.

É lógico que qualquer porta digital precisa de $V_{OL} < V_{IL}$ e de $V_{OH} > V_{IH}$. Explicando melhor, quando se vai conectar diversos dispositivos em série é interessante que se tenha as seguintes condições listadas abaixo.

- Conexão com lógica em zero: a máxima tensão de saída (V_{OL}) deve ser menor que a máxima tensão de entrada do dispositivo seguinte (V_{IL}) e
- Conexão com lógica em um: a mínima de saída (V_{OH}) deve ser maior que a mínima tensão de entrada do dispositivo seguinte (V_{IH}).

Formatado: Fonte: (Padrão) Arial

Formatado: Parágrafo da Lista, Com marcadores + Nível: 1 + Alinhado em: 0,63 cm + Recuar em: 1,27 cm

É lógico que qualquer porta digital precisa de $V_{OL} < V_{IL}$ e de $V_{OH} > V_{IH}$. Para o MSP430 F5529, que pode trabalhar com tensões na faixa de 1,8 a 3,6 V, é importante consultar o manual da versão que está sendo usada para determinar os valores exatos de tensão que definem esses níveis lógicos. Vamos exemplificar abordando o problema de realizar conexões entre o MSP (3,3 V) e portas TTL (5,0 V). A Figura 3.18 apresenta um gráfico com esses limites de tensões e níveis lógicos. Note que, para beneficiar a clareza, a figura não está em escala. Do lado esquerdo temos esses parâmetros para a família TTL e do lado direito, esses mesmos parâmetros para o MSP430 quando alimentado com 3,3 V.

Comentado [DC9]: Não ficou muito claro que as expressões se referem à conexão de dois dispositivos diferentes. Só fui entender dessa forma com a leitura dos parágrafos seguintes.

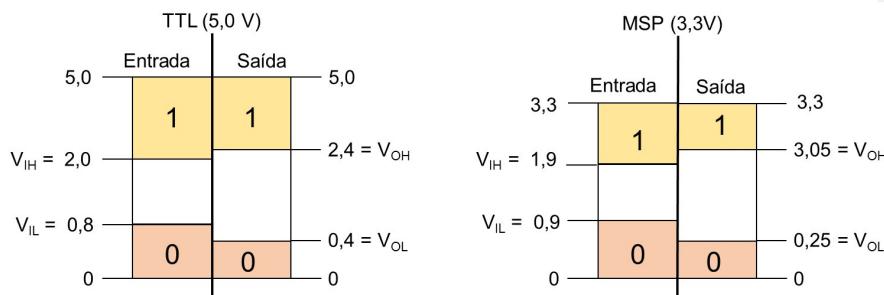


Figura 3.18. Definição dos níveis lógicos usados pela família TTL e dos níveis lógicos do MSP430 quando alimentado com 3,3 V (desenhos fora de escala).

Vejamos o que acontece se conectarmos a saída do MSP430 a uma entrada TTL. Isto está exemplificado no lado direito da Figura 3.19. Note que a faixa do lógico 1 e do lógico 0 na saída do MSP estão dentro das faixas aceitas na entrada do dispositivo TTL. De forma precisa $V_{OH} > V_{IH}$ e $V_{OL} < V_{IL}$. Além disso, a tensão máxima na saída do MSP é 3,3

V que está abaixo dos 5,0 V do TTL. Então, a conexão indicada no lado direito da figura pode ser feita!

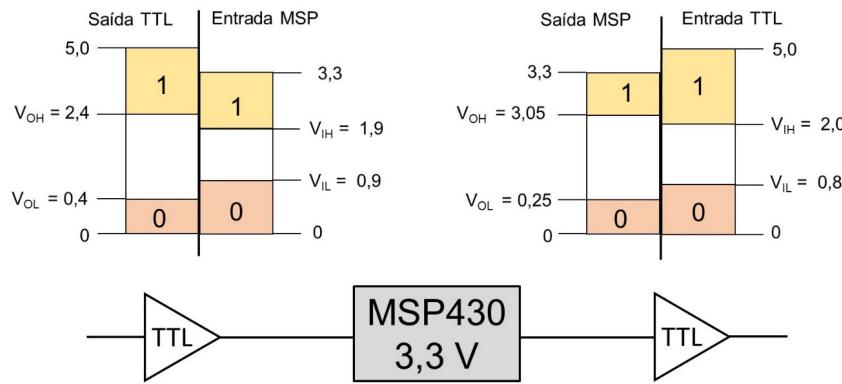


Figura 3.19. O problema de conexão entre o MSP430 (3,3 V) e portas TTL (5,0V) (desenhos fora de escala).

Analisemos agora o lado esquerdo da Figura 3.19, que ilustra o MSP recebendo a saída de um dispositivo TTL. A análise dos gráficos mostra que, novamente, $V_{OH} > V_{IH}$ e $V_{OL} < V_{IL}$. Assim, o MSP pode receber sinais TTL. Em parte, está correto, mas a tensão máxima da saída TTL é 5,0 V e o limite do MSP é 3,3 V e isto vai gerar um problema com o circuito de proteção das portas.

Como todo circuito CMOS, o MSP430 usa diodos na entrada para proteger seus pinos de tensões eletroestáticas. Isso é feito com dois diodos conectados de forma que a polarização seja reversa. A Figura 3.20 apresenta tal esquema de proteção. O diodo D1 passa a conduzir quando a tensão no pino for maior que V_{CC} mais 0,3 V ($V_P > V_{CC} + 0,3 V = 3,6 V$) e o diodo D2 passa a conduzir quando a tensão no pino estiver 0,3 V abaixo da referência de terra ($V_P < GND - 0,3 V = -0,3 V$).

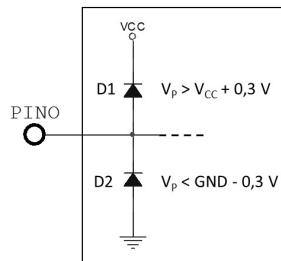


Figura 3.20. Circuito de proteção usado no GPIO do MSP.

Voltemos ao problema, que é uma saída TTL conectada a uma entrada do MSP430. Se essa saída TTL ultrapassar os 3,6 V, o diodo D1 começa a conduzir e como não existe resistor para limitar a quantidade de corrente, ela pode assumir valores elevados e “queimar” o diodo D1. Ao “queimar” um diodo, dois resultados são os mais prováveis: o diodo vira um circuito-aberto ou o diodo vira um curto-circuito. Se virar um circuito-aberto, o pino fica sem proteção contra eletricidade estática, mas ainda pode ser usado. Por outro lado, se virar um curto-circuito, o pino fica “preso” em Vcc e assim está inutilizável.

Para que o MSP430 receba sinais de dispositivos alimentados com 5,0 V, é preciso fazer a adequação dos níveis de tensão. A forma mais simples e barata de se fazer isso é usando um divisor resistivo. A Figura 3.21 apresenta uma sugestão usando dois resistores denominados de R1 e R2, formando um divisor resistivo.

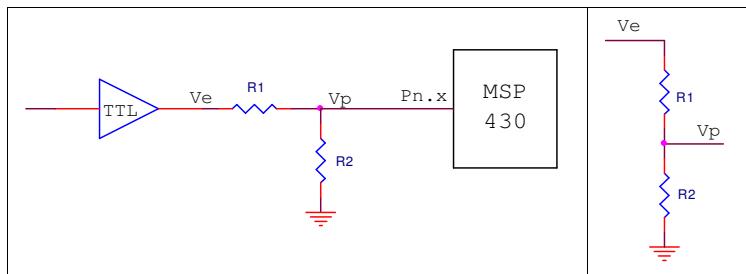


Figura 3.21. Divisor resistivo para limitar a faixa de tensão da saída TTL quando conectada a uma entrada do MSP430.

Vamos detalhar o cálculo desses dois resistores. A analisando o circuito da Figura 3.21, é fácil de constatar a validade das duas equações abaixo, onde Ve representa a tensão na entrada do divisor resistivo e Vp a tensão no pino do MSP.

$$V_p = V_e \frac{R2}{R1 + R2} \quad \frac{V_p}{V_e} = \frac{R2}{R1 + R2}$$

Vamos usar a porção da esquerda da Figura 3.19 para entendermos as três considerações importantes no cálculo desses dois resistores, que estão apresentadas na Tabela 3.3.

Tabela 3.3. Considerações para o cálculo do divisor resistivo que compatibiliza níveis de tensão de uma saída TTL com as exigências de uma entrada do MSP430

1	Quando a saída é 1, a mínima tensão na entrada (V_e) deve estar acima da mínima tensão exigida no pino (V_p), ou seja, $V_{OH,TTL} > V_{IH,MSP}$.	$\frac{V_p}{V_e} = \frac{R2}{R1 + R2} > \frac{1,9}{2,4} = 0,79$
2	Máxima tensão de saída em nível baixo precisa estar abaixo da máxima tensão de entrada em nível baixo, ou seja, $V_{OL,TTL} < V_{IL,MSP}$. Já está correto.	$(V_{OL} = 0,4) < (V_{IL} = 0,9)$ OK
3	Máxima tensão na entrada ($V_e = 5,0$ V) precisa estar abaixo do limite de proteção, que é de 3,6 V. Vamos usar 3,3 V, por segurança.	$\frac{V_p}{V_e} = \frac{R2}{R1 + R2} < \frac{3,3}{5,0} = 0,66$

A análise da tabela acima mostra que as condições 1 e 3 conflitam. Precisamos de:

$$\frac{R2}{R1 + R2} > 0,79 \quad e \quad \frac{R2}{R1 + R2} < 0,66$$

A relação precisa ser maior que 0,79 para garantir os níveis lógicos, ao mesmo tempo em que precisa ser menor que 0,66 para não acionar o diodo de proteção. Parece que não tem saída. Entretanto, temos ainda uma certa liberdade na segunda condição. O diodo de proteção, de acordo com o manual (MSP430 F5529), suporta uma corrente de até 2 mA. Assim, não há problema em acionar esse diodo, desde que a corrente não ultrapasse o limite de 2 mA. Vamos então calcular os resistores usando a condição 1 e depois verificar a corrente pelo resistor. Manipulando a equação da condição 1, podemos explicitar $R1$ em função de $R2$, como mostrado abaixo.

$$R1 < \frac{0,21}{0,79} R2$$

Se, por simplicidade, escolhermos $R2 = 10 \text{ k}\Omega$, temos $R1 < 2,65 \text{ k}\Omega$. O valor comercial próximo é $2,4 \text{ k}\Omega$. Sob essas condições, certamente vamos acionar o diodo de proteção (D1). Mas, qual o valor da corrente que vai passar pelo diodo? De forma bem simplificada, considerando que na condução o diodo é um curto, temos

$$I = \frac{5 - 3,3}{2,4 \text{ k}} = 0,7 \text{ mA}$$

que é um valor perfeitamente tolerável pelo diodo de proteção. Com as três perguntas abaixo, convidamos o leitor a trabalhar um pouco mais este conceito.

Pergunta 1) É muito comum encontrarmos na Internet circuitos de compatibilização usando os seguintes valores: $R1 = 4,7 \text{ k}\Omega$ e $R2 = 10 \text{ k}\Omega$. Quais os níveis de tensão resultantes? Há algum risco para os níveis lógicos? Qual a máxima corrente que passa pelo diodo de proteção?

Pergunta 2) Há alguma vantagem em aumentar o valor de $R1$, por exemplo, escolher $R1 = 100 \text{ k}\Omega$?

Pergunta 3) E se o MSP430 F5529 fosse alimentado com 2,0 V? Consulte o manual do F5529 para ver a relação entre as tensões e os níveis lógicos. Sugira um par de resistores ($R1$ e $R2$) e calcule a corrente pelo diodo de proteção.

Para finalizar o assunto, o mercado oferece uma grande variedade de “conversores de nível lógico” para 5/3,3 V. Estão disponíveis conversores unidirecionais e conversores bidirecionais. Eles funcionam bem, mas a experiência com alguns demonstrou que são lentos.

3.5. Mapeamento da Porta P4

A porta P4 possui um recurso especial que é muito útil em diversas situações. Todos seus bits podem ser mapeados, como saída ou entrada, para usar uma série de outros recursos. Esse mapeamento é feito com o uso do registrador P4MAPx, onde “x” representa o bit de P4 a ser mapeado. Então, veja o leitor que existe um registrador para cada bit da porta P4. O registrador P4MAP5, faz o mapeamento do bit 5 da porta P4. A

Tabela 3.2, que é cópia da Tabela 10 do manual do MSP430 apresenta as possibilidades de mapeamento da porta P4. Por enquanto, os elementos desta tabela podem parecer incompreensíveis, mas muitos serão abordados nos próximos capítulos.

Tabela 3.4. Cópia da Tabela 10 do Manual do MSP430, apresentando os recursos de mapeamento da porta P4 (registraror P4MAP)

Table 10. Port Mapping Mnemonics and Functions

VALUE	PxMAPy MNEMONIC	INPUT PIN FUNCTION	OUTPUT PIN FUNCTION
0	PM_NONE	None	DVSS
1	PM_CBOUTO	-	Comparator_B output
	PM_TB0CLK	TB0 clock input	
2	PM_ADC12CLK	-	ADC12CLK
	PM_DMAE0	DMAE0 input	
3	PM_SVMOUT	-	SVM output
	PM_TB0OUTH	TB0 high impedance input TB0OUTH	
4	PM_TB0CCR0A	TB0 CCR0 capture input CCI0A	TB0 CCR0 compare output Out0
5	PM_TB0CCR1A	TB0 CCR1 capture input CCI1A	TB0 CCR1 compare output Out1
6	PM_TB0CCR2A	TB0 CCR2 capture input CCI2A	TB0 CCR2 compare output Out2
7	PM_TB0CCR3A	TB0 CCR3 capture input CCI3A	TB0 CCR3 compare output Out3
8	PM_TB0CCR4A	TB0 CCR4 capture input CCI4A	TB0 CCR4 compare output Out4
9	PM_TB0CCR5A	TB0 CCR5 capture input CCI5A	TB0 CCR5 compare output Out5
10	PM_TB0CCR6A	TB0 CCR6 capture input CCI6A	TB0 CCR6 compare output Out6
11	PM_UCA1RXD	USCI_A1 UART RXD (Direction controlled by USCI - input)	
	PM_UCA1SOMI	USCI_A1 SPI slave out master in (direction controlled by USCI)	
12	PM_UCA1TXD	USCI_A1 UART TXD (Direction controlled by USCI - output)	
	PM_UCA1SIMO	USCI_A1 SPI slave in master out (direction controlled by USCI)	
13	PM_UCA1CLK	USCI_A1 clock input/output (direction controlled by USCI)	
	PM_UCB1STE	USCI_B1 SPI slave transmit enable (direction controlled by USCI)	
14	PM_UCB1SOMI	USCI_B1 SPI slave out master in (direction controlled by USCI)	
	PM_UCB1SCL	USCI_B1 I2C clock (open drain and direction controlled by USCI)	
15	PM_UCB1SIMO	USCI_B1 SPI slave in master out (direction controlled by USCI)	
	PM_UCB1SDA	USCI_B1 I2C data (open drain and direction controlled by USCI)	
16	PM_UCB1CLK	USCI_B1 clock input/output (direction controlled by USCI)	
	PM_UCA1STE	USCI_A1 SPI slave transmit enable (direction controlled by USCI)	
17	PM_CBOUT1	None	Comparator_B output
18	PM_MCLK	None	MCLK
19 - 30	Reserved	None	DVSS
31 (0FFh) ⁽¹⁾	PM_ANALOG	Disables the output driver as well as the input Schmitt-trigger to prevent parasitic cross currents when applying analog signals.	

O mapeamento é muito simples, mas para evitar que ele ocorra acidentalmente, o fabricante disponibilizou uma chave—por software ([acesso protegido por senha](#)). Isto significa que, antes de fazer qualquer mapeamento de P4, é preciso habilitar essa chave. Uma vez habilitada a chave, ela permanece assim por 32 ciclos (períodos de MCLK) e depois volta a trancar o mapeamento. O exemplo a seguir ilustra o mapeamento de P4.7 para dar saída ao comparador 0 do timer B0 (TB0CCR0A – TB0 CCR0 compare output 0).

Exemplo do mapeamento do bit P4.7

BIS.B #BIT7,&P4DIR	; P4.7 = saída
BIS.B #BIT7,&P4SEL	; P4.7 = função alternativa
MOV #0x02D52,&PMAPKEYID	; Chave para liberar mapeamento
MOV #PM_TB0CCR0A,&P4MAP7	; Mapeamento desejado

3.6. GPIO com a LaunchPad F5529

O *kit* usado no laboratório é a LaunchPad F5529. Essa pequena placa facilita o uso de diversos pinos de I/O e ainda oferece dois botões e dois *leds*. Note que nem todas as portas de GPIO estão disponíveis nas barras de pinos. A Figuras 3.22 e 3.23 apresentam os recursos disponíveis neste LaunchPad e a Tabela 3.1 apresenta esses recursos ordenados segundo as portas.

Tabela 3.5. Os pinos disponíveis na LaunchPad estão marcados com “”*

	7	6	5	4	3	2	1	0	Total
P0	-	-	-	-	-	-	-	-	0
P1	-	*	*	*	*	*	-	-	5
P2	*	*	*	*	*	*	-	*	6
P3	*	*	*	*	-	*	*	*	6
P4	-	-	-	-	*	*	*	*	4
P5	-	-	-	-	-	-	-	-	0
P6	-	*	*	*	*	*	*	*	6
P7	-	-	-	*	-	-	-	*	2
P8	-	-	-	-	-	-	*	-	1

← Tabela formatada

← Tabela formatada

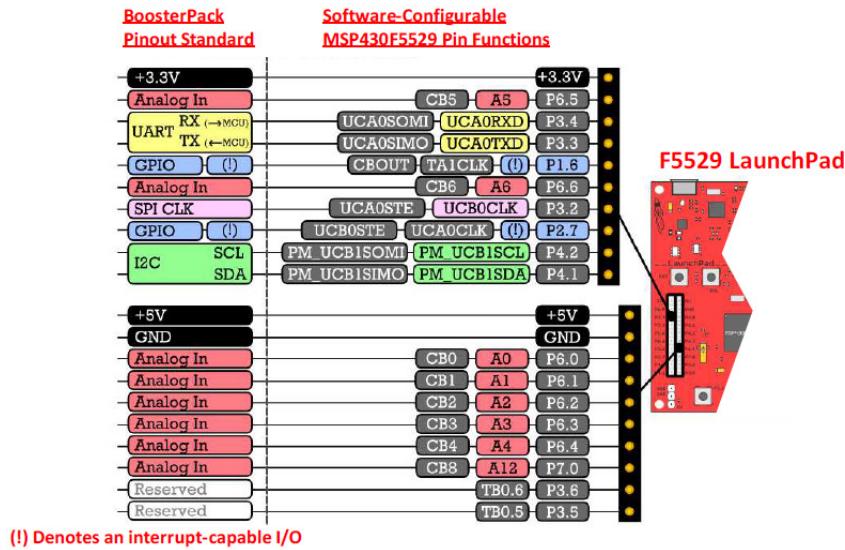
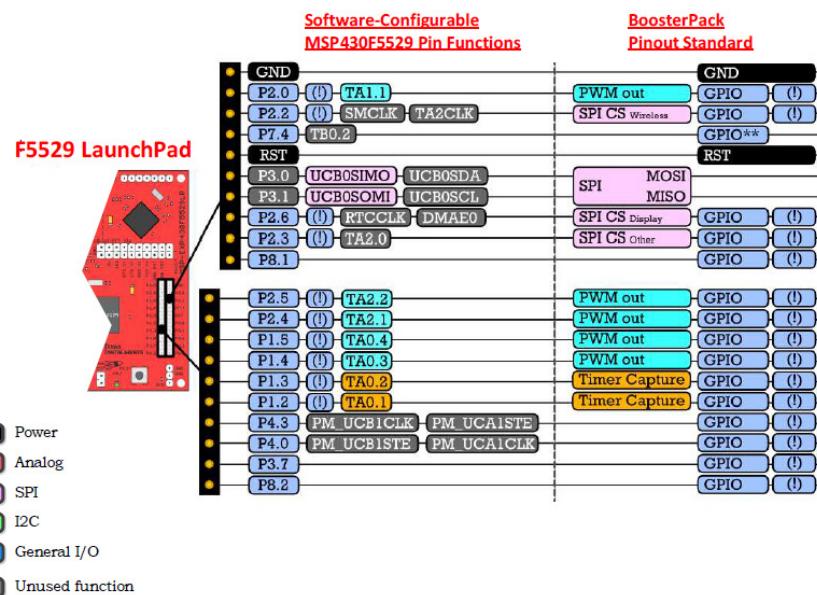


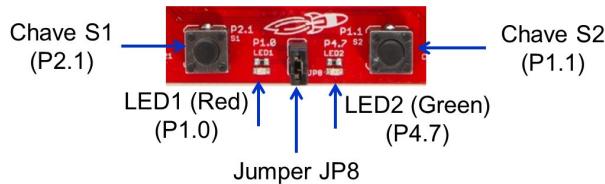
Figura 3.22. Disponibilidade de GPIO na barra de pinos do lado esquerdo da LaunchPad (Imagem copiada do manual da LaunchPad).



*Figura 3.23. Disponibilidade de GPIO na barra de pinos do lado direito da LaunchPad
(Imagem copiada do manual do LaunchPad).*

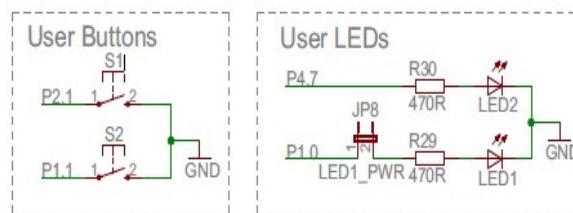
A LaunchPad ainda disponibiliza duas chaves e dois *leds*, listados abaixo. A Figura 3.24 apresenta a disposição dessas chaves e *leds*. Deve ser notado que o *led* verde (P4.7), é um bit da porta P4 e por isso, pode ser mapeado para diversas outras funções. Este recurso será usado nos próximos capítulos.

Chave S1 → P2.1	LED1 → P1.0 (vermelho)
Chave S2 → P1.1	LED2 → P4.7 (verde)



*Figura 3.24. Disposição dos leds e chaves da LaunchPad
(Imagem copiada do manual da LaunchPad).*

A Figura 3.25 apresenta o circuito de conexão das chaves e dos *leds* disponíveis na LaunchPad. Veja que o circuito é extremamente simples. As chaves, quando pressionadas, fazem curto para o terra. O que deve ser ressaltado é a existência do *jumper* JP8 no circuito que aciona o LED1. Isto significa que quando removemos o *jumper*, podemos usar um cabo conectado ao pino 2 de JP8 para receber qualquer outro sinal e comandar o LED1.



*Figura 3.25. Circuito de conexão das chaves e dos leds do LaunchPad
(Imagem copiada do manual do LaunchPad).*

Cada *led* está em série com um resistor de $470\ \Omega$ que serve para limitar sua corrente. Supondo que em nível alto a tensão no pino seja de 3,3 V e que a queda de tensão sobre o *led* seja de 1,8 V, a corrente será, aproximadamente de 3,2 mA, como calculado abaixo.

$$I = \frac{V}{R} = \frac{3,3 - 1,8}{470} = 3,2\ mA$$

Qual o limite de corrente de cada pino?

Observação: de acordo com o manual do MSP430F5529, a máxima corrente que pode ser fornecida por um pino é de 5 mA. Se o driver de alto desempenho (DS=1) for usado, ela sobe para 15 mA.

Comentado [DC10]: 5mA sem DS e 15mA c/ DS (encontrei no datasheet)

3.7. Registradores das Portas de I/O

A seguir são apresentadas figuras dos diversos registradores usados pelas portas de I/O. Eles são, de certa forma, monótonos, já que seus bits não possuem nomes individuais. Para tornar completa a descrição, apesar de ser assunto do próximo capítulo, aqui também serão apresentados os registradores relacionados com as interrupções das portas P1 e P2.

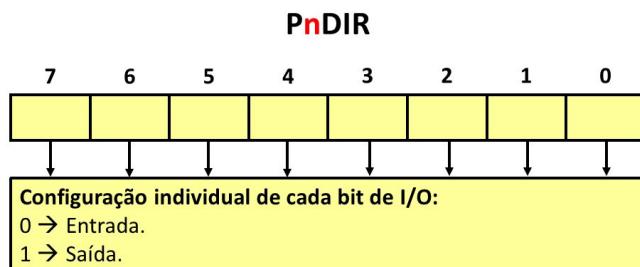


Figura 3.26. Registrador para controlar a direção dos bits da porta de I/O.

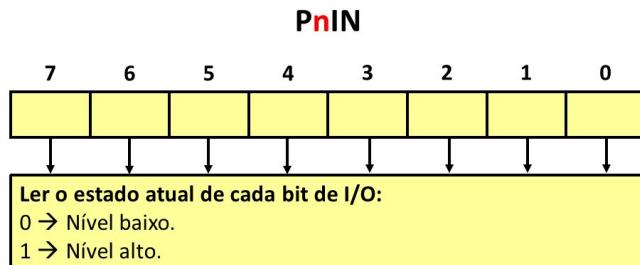


Figura 3.27. Registrador para realizar a leitura dos bits da porta de I/O.

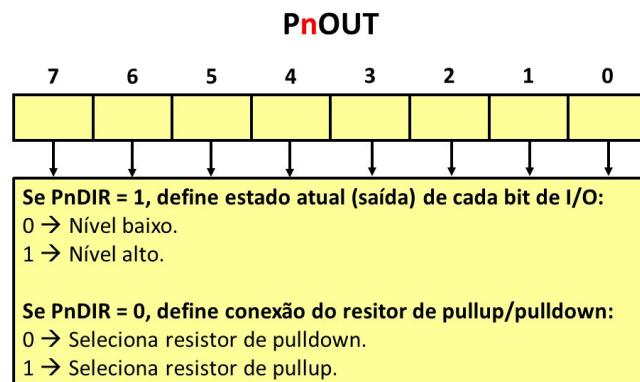
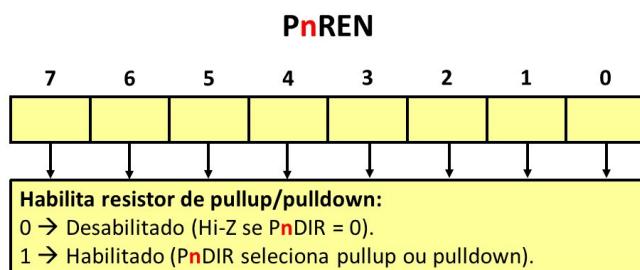
Figura 3.28. Registrador para controlar o estado da saída de I/O ($PnDIR = 1$) ou para selecionar resistor de pullup/pulldown ($PnDIR = 0$).

Figura 3.29. Registrador para habilitar ou desabilitar o uso dos resistores de pullup ou pulldown.

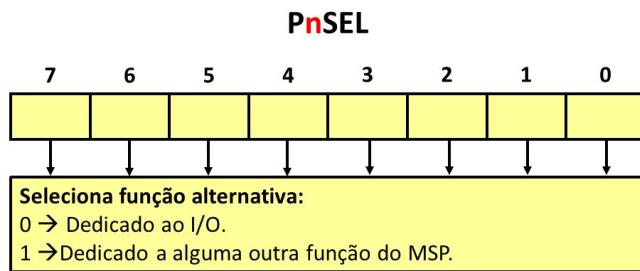


Figura 3.30. Registrador para indicar se determinado bit de I/O será usado como GPIO ou se será dedicado a uma determinada função do MSP.

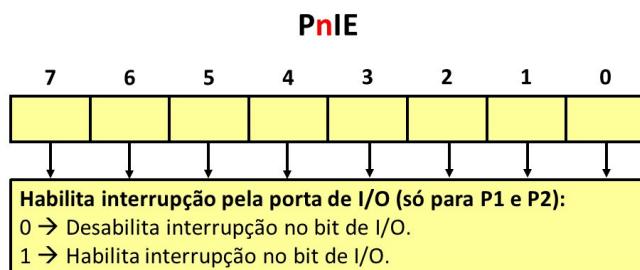


Figura 3.31. Registrador para habilitar ou desabilitar a interrupções por um bit de GPIO.
Válido apenas para as portas P1 e P2.

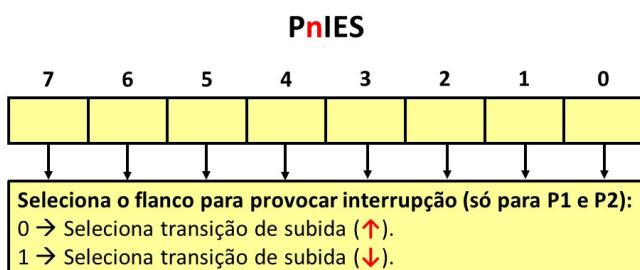


Figura 3.32. Registrador para indicar o tipo de flanco que prova interrupção. Válido apenas para as portas P1 e P2.

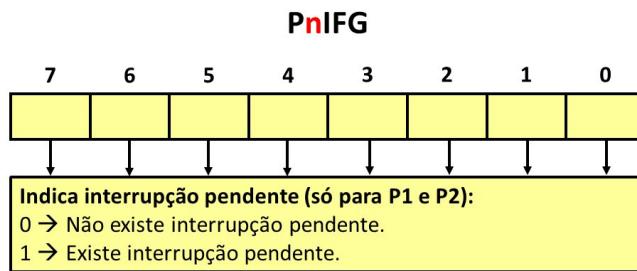


Figura 3.33. Registrador de flags de interrupção. Indica quais interrupções estão pendentes. Válido apenas para as portas P1 e P2.

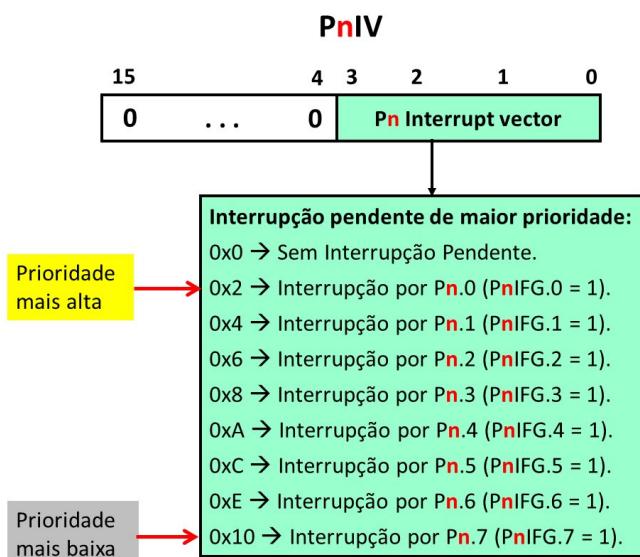


Figura 3.34. Registrador para indicar a interrupção pendente de maior prioridade. Válido apenas para as portas P1 e P2.

3.8. Exercícios Resolvidos

A seguir são apresentados diversos exercício resolvidos. O leitor é convidado a executar todos esses exercícios usando o Code Composer Studio (CCS) e a placa **MSP430 F5529 LaunchPad**. Nas soluções apresentadas, para evitar a extensão do programa, colocamos na mesma listagem a sub-rotina pedida e o ambiente para seu teste.

De nada adianta simplesmente ler os problemas e as soluções. Seria como tentar aprender a andar de bicicleta apenas observando os ciclistas no parque. É preciso que o leitor tente esboçar sua solução e depois a compare com a solução apresentada. É necessário desenvolver a habilidade de *pensar com os recursos de GPIO*.

ER 3.1. Escreva a sub-rotina P_VM que fica presa em um laço, piscando o LED1 (P1.0) que é o vermelho. É questionável se algo que fica preso em um laço infinito é realmente uma sub-rotina. Vamos considerar que sim.

Solução:

Precisamos configurar o pino P1.0 como saída e ficar alterando-o entre os estados 0 e 1. Como o processador é muito rápido, precisamos garantir que a saída fique algum tempo em cada um desses estados, para que o programador possa vê-los. Isto vai ser conseguido pela sub-rotina TEMPO que, como o nome diz, apenas consome tempo. A listagem abaixo apresenta uma solução.

Listagem do ER 3.1, versão a

```
;ER 3.1.a
TP      .equ  65535          ;Valor para o atraso (chute)
;
CALL  #P_VM            ;Chamar sub-rotina
JMP   $              ;Prender MSP
;
; Sub-rotina para piscar led vermelho (P1.0)
P_VM:    BIS.B #BIT0,&P1DIR      ;Configurar P1.0 = saída
P_VM1:   BIS.B #BIT0,&P1OUT       ;P1.0 = 1 (acende)
        CALL  #TEMPO           ;Atraso
        BIC.B #BIT0,&P1OUT       ;P1.0 = 0 (apaga)
        CALL  #TEMPO           ;Atraso
        JMP   P_VM1            ;Repetir laço
;
; Sub-rotina para consumir tempo
TEMPO:   MOV   #TP,R5          ;Inicializar contador
```

```

TEMP1      DEC    R5          ;Decrementar contador
          JNZ    TEMP1        ;Retornar quando chegar a zero
          RET

```

Existe uma forma mais compacta de escrever o programa solução. A operação OU-Exclusivo pode ser usada para inverter o estado de um *bit*. Ficamos então em um laço: inverte saída, chama atraso e repete tudo. Veja a listagem a seguir.

Listagem do ER 3.1, versão b

```

;ER 3.1.b
TP      .equ  65535       ;Valor para o atraso (chute)

        CALL  #P_VM        ;Chamar sub-rotina
        JMP   $              ;Prender MSP
;

; Sub-rotina para piscar led vermelho (P1.0)
P_VM:   BIS.B #BIT0,&P1DIR    ;Configurar P1.0 = saída
P_VM1:  XOR.B #BIT0,&P1OUT    ;Inverter P1.0
        CALL  #TEMPO        ;Atraso
        JMP   P_VM1         ;Repetir laço
;

; Sub-rotina para consumir tempo
TEMPO:  MOV   #TP,R5        ;Inicializar contador
TEMP1:  DEC   R5          ;Decrementar contador
          JNZ    TEMP1        ;Retornar quando chegar a zero
          RET

```

É importante notar que construímos a rotina que gera o atraso com um laço de programa. Por enquanto, vamos usar este recurso, porém ele é uma solução muito pobre, pois além de desperdiçar a CPU decrementando um contador, torna difícil o ajuste do valor do atraso. Para piorar, se o relógio da CPU for alterado, o valor do atraso muda.

Medindo com um osciloscópio, o período de acionamento do LED1 ficou igual a 371,2 ms (185,6 ms aceso e 185,6 ms apagado), o que corresponde a uma frequência aproximada de 2,69 Hz. E se quiséssemos aumentar a frequência para 5 Hz? É possível calcular. A conta (uma regra de três) abaixo nos mostra que esse valor seria 35.258. O leitor é convidado a testá-lo.

$$TP = \frac{2,69}{5} 65535 = 35258$$

Mais adiante, quando estudarmos os *timers*, veremos como fazer isso com precisão. Por hora, essa é a alternativa que temos. Lembrando que o maior valor que se consegue representar com 16 bits é 65.535, então nossa sub-rotina TEMPO já está com o atraso máximo. Podemos aumentar a frequência do piscar, mas não conseguimos ir abaixo de 2,69 Hz. Podemos adicionar instruções NOP na sub-rotina TEMPO, e assim, de forma bruta, aumentarmos o atraso. Veja então que encontramos uma utilidade para a instrução NOP! De acordo com a listagem abaixo, adicionamos 2 NOPs na sub-rotina TEMPO. Neste caso, usando um osciloscópio, o período medido foi de 618,7 ms (309,4 ms aceso e 309,46 ms apagado), o que corresponde a 1,62 Hz.

Listagem da nova versão da sub-rotina TEMPO

```
; Sub-rotina para consumir mais tempo
TEMPO:    MOV    #TP,R5           ; Inicializar contador
TEMP1:    NOP                ; Nada
          NOP                ; Nada
          DEC    R5           ; Decrementar contador
          JNZ    TEMP1         ; Retornar quando chegar a zero
          RET
```

ER 3.2. Vamos aproveitar o ambiente do exercício anterior para trabalharmos um conceito importante, que trata do tempo gasto por cada instrução. Sabendo que o processador vem configurado para trabalhar com $MCLK = 2^{20} = 1.048.576$ Hz, e de posse da Tabela 3.7, que indica o número de ciclos gasto por cada instrução, calcule a duração de ambas sub-rotinas TEMPO (primeira versão e a segunda, que usa NOP).

Tabela 3.7. Tempo gasto pelas instruções da sub-rotina TEMPO.

Instrução	Ciclos
MOV #TP, R5	2
DEC R5	1
JNZ TEMP1	2
RET	2
NOP	1

Solução:

O ciclo citado na Tabela 3.6, corresponde a um período relógio da CPU que é o MCLK. Assim que “sai da caixa”, o LaunchPad tem seu MCLK configurado para 1.048.576 Hz (2^{20} Hz). Esse é o relógio que usaremos como referência. A vamos então analisar a rotina para ver quando gasta cada instrução.

```
; Sub-rotina para consumir tempo
TEMPO:    MOV    #TP,R5           ;2 ciclos
TEMP1:    DEC    R5             ;TP x 1 ciclo
          JNZ    TEMP1           ;TP x 2 ciclos
          RET                 ;2 ciclos
```

- O total de ciclos é dado por: $2 + 1 \cdot TP + 2 \cdot TP + 2 = 4 + 3 \cdot TP = 196.609$ ciclos.
- Sendo assim, o tempo gasto é: $196.609 / 1.048.576 = 187,5$ ms.
- Com o osciloscópio foi medido 185,6 ms, um valor bem próximo ao calculado.

Para o caso em que se usam NOPs na sub-rotina, temos:

```
; Sub-rotina para consumir tempo
TEMPO:    MOV    #TP,R5           ;2 ciclos
TEMP1:    NOP                 ;1 ciclo
          NOP                 ;1 ciclo
          DEC    R5             ;1 ciclo
          JNZ    TEMP1           ;2 ciclos
          RET                 ;2 ciclos
```

- O total de ciclos é dado por: $4 + 5 \cdot TP = 327.675$ ciclos.
- Sendo assim, o tempo gasto é $327.675 / 1.048.576 = 312,5$ ms.
- Com o osciloscópio foi medido 309,4 ms, um valor bem próximo ao calculado.

ER 3.3. Vamos aproveitar um pouco mais esse ambiente de atrasos para propor outro problema. Queremos que o pisca-pisca do LED1 seja de 1 Hz (0,5 s aceso e 0,5 s apagado). Lembre-se de que o valor máximo que conseguimos representar com 16 bits é 65.535. Repetimos que, mais adiante, quando estudarmos os *timers*, veremos como gerar intervalos de tempo com grande precisão.

Solução:

Poderíamos tentar a mesma solução adicionando mais um NOP e alterando o valor de TP. Porém, queremos apresentar uma outra forma, que é o aninhamento de contadores, como mostrado abaixo. O contador externo é feito com R6 e o contador interno com R5. Cada vez que R5 completa seu ciclo, decrementa R6 uma vez e é recarregado. Enquanto R6 for diferente de zero, repete-se o ciclo do contador com R5.

```
TP1      .equ 100
TP2      .equ 5000
;
; Sub-rotina para consumir tempo
```

```

TEMPO:    MOV    #TP2,R6           ;Inicializar contador externo
TEMP1:    MOV    #TP1,R5           ;Inicializar contador interno
TEMP2:    DEC    R5              ;Decrementar interno
          JNZ    TEMP2          ;Laço interno até chegar a zero
          DEC    R6              ;Decrementar externo
          JNZ    TEMP1          ;Laço externo até chegar a zero
          RET

```

Repetimos a solução do ER 3.1 e procuramos determinar valores para TP1 e TP2 de forma que se consiga piscar o *led* em 0,5 Hz. Os valores indicados na listagem acima resultaram 2,88 s, medidos com a ajuda de um osciloscópio. Vamos tentar achar uma combinação que resulte em 1 Hz. Uma forma rápida é manter o valor de TP1 em 100 e calcular TP2 usando regra de três, como mostrado abaixo.

$$TP2 = \frac{1,0}{2,88} 5000 = 1736,11$$

ER 3.4. Escreva a sub-rotina P_VM_VD que fica presa em um laço, piscando o LED1 (P1.0), que é o vermelho, e o LED2 (P4.7), que é o verde, na frequência de 1 Hz (0,5 s aceso e 0,5 s apagado), mas de forma complementar. Use os resultados do ER 3.3 para gerar os atrasos.

Solução:

Precisamos configurar os pinos P1.0 (LED1) e P4.7 (LED2) como saída, inicializá-los de forma complementar e depois ficar alterando o estado de cada um. No exercício anterior já calculamos os valores dos atrasos. A listagem abaixo apresenta uma solução.

Listagem do ER 3.4

```

;ER 3.4
TP1      .equ  100
TP2      .equ  1736
;
        CALL  #P_VM_VD          ;Chamar sub-rotina
        JMP   $                  ;Prender MSP
;
; Sub-rotina para piscar led vermelho (P1.0)
P_VM_VD:  BIS.B #BIT0,&P1DIR      ;Configurar P1.0 = saída
          BIS.B #BIT0,&P1OUT      ;P1.0 = 1 - aceso
          BIS.B #BIT7,&P4DIR      ;Configurar P4.7 = saída
          BIC.B #BIT4,&P4OUT      ;P4.7 = 0 - apagado
P_VMVD1:  XOR.B #BIT0,&P1OUT      ;Inverter P1.0
          XOR.B #BIT7,&P4OUT      ;Inverter P4.7

```

```

        CALL  #TEMPO          ;Atraso
        JMP   P_VMV1           ;Repetir laço
;
; Sub-rotina para consumir tempo
TEMPO:    MOV   #TP2,R6      ;Inicializar contador externo
TEMP1:    MOV   #TP1,R5      ;Inicializar contador interno
TEMP2:    DEC   R5          ;Decrementar interno
          JNZ   TEMP2         ;Laço interno até chegar a zero
          DEC   R6          ;Decrementar externo
          JNZ   TEMP1         ;Laço externo até chegar a zero
          RET

```

ER 3.5. Escreva a sub-rotina P_SW_VM que mantém aceso o LED1 (P1.0), que é o vermelho, enquanto a chave S1 (P2.1) permanecer acionada.

Solução:

Precisamos configurar o pino P1.0 (LED1) como saída e configurar o pino P2.1 (S1) como entrada. Para que a entrada do pino não fique em alta impedância quando a chave estiver aberta, vamos usar o resistor de *pullup*. Observe na Figura 3.25 que a chave fecha um curto para terra quando acionada. Assim, só podemos usar o *pullup*.

A rotina é simples e fica presa em dois laços:

- Enquanto S1 está aberta, fica apagando o LED1 e
- Enquanto S1 está fechada, fica acendendo o LED1.

Listagem da solução do ER 3.5

```

;ER 3.5
        CALL  #P_SW_VM          ;Chamar sub-rotina
        JMP   $                  ;Prender MSP
;
; Sub-rotina : acender LED1 (P1.0) com o acionamento de S1 (P2.1)
P_SW_VM:  BIS.B #BIT0,&P1DIR      ;Configurar P1.0 = saída
          BIC.B #BIT0,&P1OUT      ;P1.0 = 0 - LED1 apagado
          ;
          BIC.B #BIT1,&P2DIR      ;Configurar P2.1 = entrada
          BIS.B #BIT1,&P2REN       ;Habilitar resistor
          BIS.B #BIT1,&P2OUT       ;Selecionar pullup
          ;
LB1:     BIC.B #BIT0,&P1OUT      ;P1.0 = 0 - LED1 apagado
          BIT.B #BIT1,&P2IN       ;Testar P2.1 = 1?
          JNZ   LB1                ;Aberto, voltar ao LB1
          ;
LB2:     BIS.B #BIT0,&P1OUT      ;P1.0 = 1 - LED1 aceso
          BIT.B #BIT1,&P2IN       ;Testar P2.1 = 1?
          JZ    LB2                ;Fechado, voltar ao LB2
          JMP   LB1                ;Se aberto, voltar

```

ER 3.6. Escreva a sub-rotina P_SL que mantém aceso o LED1 (P1.0), que é o vermelho, enquanto a chave S1 (P2.1) permanecer acionada e o LED2 (P4.7), que é o verde, enquanto a chave S2 (P1.1) permanecer acionada.

Solução:

Precisamos fazer as configurações listadas abaixo.

P1.0 (LED1) → saída	P4.7 (LED2) → saída
P2.1 (S1) → entrada com <i>pullup</i>	P1.1(S2) → entrada com <i>pullup</i> .

O fluxograma apresentado na Figura 3.26 ilustra a solução. Ele é bem simples e, de acordo com o acionamento das chaves, acende ou apaga os *leds* correspondentes. A listagem do programa está apresentada logo a seguir. Veja que, para facilitar a escrituração do programa criamos a sub-rotina CONFIG, só para fazer as configurações das portas.

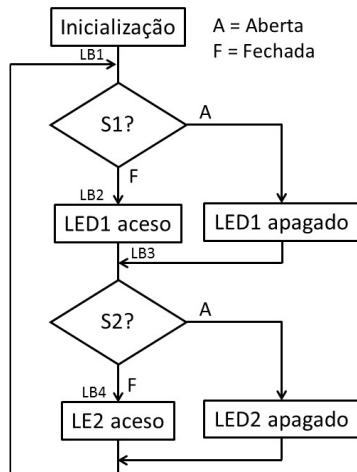


Figura 3.26. Fluxograma do ER 3.6, para as duas chaves (S1 e S2) comandarem os dois leds (LED1 e LED2).

Listagem da solução do ER 3.6

```

;ER 3.6
        CALL  #P_SL           ;Chamar sub-rotina
        JMP   $                 ;Prender MSP
;

```

```

; Sub-rotina : acender LED1 (P1.0) com o acionamento de S1 (P2.1)
P_SL:      CALL  #CONFIG           ;Configurar pinos
LB1:      BIT.B #BIT1,&P2IN        ;Testar S1
          JZ    LB2             ;Se acionada vai para LB2
          BIC.B #BIT0,&P1OUT       ;P1.0 = 0 - LED1 apagado
          JMP   LB3             ;Ir para próxima chave
LB2:      BIS.B #BIT0,&P1OUT       ;P1.0 = 0 - LED1 aceso
          ;
LB3:      BIT.B #BIT1,&P1IN        ;Testar S2
          JZ    LB4             ;Se acionada vai para LB4
          BIC.B #BIT7,&P4OUT       ;P4.7 = 0 - LED2 apagado
          JMP   LB1             ;Repetir o laço
LB4:      BIS.B #BIT7,&P4OUT       ;P4.7 = 1 - LED2 aceso
          JMP   LB1             ;Repetir o laço
;
; Configurar os pinos
CONFIG:   BIS.B #BIT0,&P1DIR       ;Configurar P1.0 = saída
          BIC.B #BIT0,&P1OUT       ;P1.0 = 0 - LED1 apagado
          ;
          BIS.B #BIT7,&P4DIR       ;Configurar P4.7 = saída
          BIC.B #BIT7,&P4OUT       ;P4.7 = 0 - LED2 apagado
          ;
          BIC.B #BIT1,&P2DIR       ;Configurar P2.1 = entrada
          BIS.B #BIT1,&P2REN        ;Habilitar resistor
          BIS.B #BIT1,&P2OUT        ;Selecionar pullup
          ;
          BIC.B #BIT1,&P1DIR       ;Configurar P1.1 = entrada
          BIS.B #BIT1,&P1REN        ;Habilitar resistor
          BIS.B #BIT1,&P1OUT        ;Selecionar pullup
          RET

```

ER 3.7. Escreva a sub-rotina C_SL que usa a chave S1 (P2.1) para inverter o estado do LED1 (P1.0), que é o vermelho. A cada acionamento de S1 (transição do estado aberta para fechada), o estado do LED1 é invertido.

Solução:

Precisamos fazer as configurações listadas abaixo.

P1.0 (LED1) → saída P2.1 (S1) → entrada com *pullup*

Como primeira abordagem, apresentamos o fluxograma da Figura 3.27. À primeira vista o fluxograma parece correto. Entretanto, ele tem um grave defeito. O leitor consegue identificar o erro deste fluxograma?

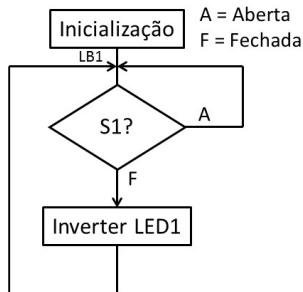


Figura 3.27. Fluxograma com uma sugestão para solução (defeituosa) do ER 3.7, que pede para inverter o estado do LED1 a cada acionamento da chave S1.

Mesmo sabendo que o fluxograma tem erro, vamos escrever o programa correspondente e executá-lo. Durante sua execução tente identificar o que está errado. A listagem abaixo traz o programa em questão.

Listagem da solução do ER 3.7, versão a (com problemas)

```

;ER 3.7.a (defeituosa)
        CALL  #C_SL           ;Chamar sub-rotina
        JMP   $               ;Prender MSP
;
; Sub-rotina : inverter LED1 (P1.0) a cada o acionamento de S1 (P2.1)
C_SL:      CALL  #CONFIG          ;Configurar pinos
LB1:      BIT.B #BIT1,&P2IN       ;Testar S1
        JNZ   LB1             ;Se aberta, voltar
        XOR.B #BIT0,&P1OUT     ;S1 Fechada: Inverter P1.0 = 0 - LED1
        JMP   LB1             ;Repetir o laço
;
; Configurar os pinos
CONFIG:    BIS.B #BIT0,&P1DIR      ;Configurar P1.0 = saída
        BIC.B #BIT0,&P1OUT      ;P1.0 = 0 - LED1 apagado
        ;
        BIC.B #BIT1,&P2DIR      ;Configurar S1 = P2.1 = entrada
        BIS.B #BIT1,&P2REN       ;Habilitar resistor
        BIS.B #BIT1,&P2OUT       ;Selecionar pullup
        RET

```

O leitor deve ter notado que enquanto a chave está aberta, o estado do *LED1* permanece estático. Porém, ao acionar a chave, não necessariamente o *led* muda de estado. Parece algo aleatório. Pode-se notar que (talvez) o brilho do *led* diminua enquanto a chave está acionada. Uma experimentação vai mostrar que o *led* fica aceso enquanto a chave S1 estiver acionada. Na verdade, ele não está aceso, mas sim acendendo e apagando numa velocidade muito alta, o que nos parece aceso.

O leitor já deve desconfiar do erro na sub-rotina. Devemos inverter o estado do LED1 quando a chave passar do estado aberto para o fechado, e nada deve feito enquanto a chave estiver acionada. Nada se faz quando ela passa do estado de fechada para aberta. Em resumo, a sub-rotina precisa esperar o usuário liberar a chave. O fluxograma abaixo se aproxima da solução correta, mas ainda tem uma pequena deficiência que será tratada no próximo exercício.

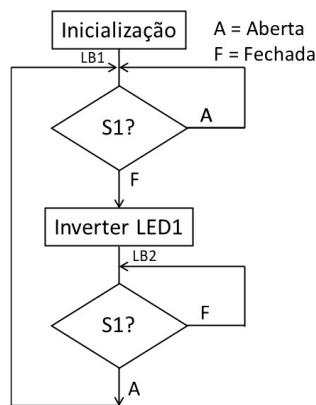


Figura 3.28. Fluxograma do ER 3.7, para inverter o estado do LED1 a cada acionamento das chave S.

A sub-rotina abaixo funciona bem melhor, porém o leitor ainda vai notar uma pequena incerteza no resultado. Algumas vezes, ao acionar a chave o led inverte de estado e outras vezes não. Isso tem a ver com os rebotes da chave, assunto que é abordado nos dois próximos problemas.

Listagem da solução do ER 3.7, versão b

```

;ER 3.7.b (correta)
    CALL  #C_SL           ;Chamar sub-rotina
    JMP   $               ;Prender MSP
;
; Sub-rotina : inverter LED1 (P1.0) a cada o acionamento de S1 (P2.1)
C_SL:   CALL  #CONFIG      ;Configurar pinos
LB1:    BIT.B #BIT1,&P2IN   ;Testar S1
        JNZ   LB1          ;Se aberta, voltar
        XOR.B #BIT0,&P1OUT   ;S1 Fechada: Inverter P1.0 = 0 - LED1
LB2:    BIT.B #BIT1,&P2IN   ;Testar S1
        JZ    LB2          ;Se fechada, repetir teste
        JMP   LB1          ;Se aberta, repetir o laço
  
```

```

;
; Configurar os pinos
CONFIG:    BIS.B #BIT0,&P1DIR      ;Configurar P1.0 = saída
            BIC.B #BIT0,&P1OUT      ;P1.0 = 0 - LED1 apagado
            ;
            BIC.B #BIT1,&P2DIR      ;Configurar S1 = P2.1 = entrada
            BIS.B #BIT1,&P2REN      ;Habilitar resistor
            BIS.B #BIT1,&P2OUT      ;Selecionar pullup
            RET                      ;Retornar

```

ER 3.8. Escreva a sub-rotina CT_SL que implementa um contador binário com os dois *leds* da placa. A cada acionamento de S1 (transição de aberta para fechada), o contador avança uma contagem, de acordo com a referência indicada na tabela abaixo.

Tabela 3.7. Contador de 2 bits construído com LED1 e LED2.

Contagem	LED1	LED2
0	Apagado	Apagado
1	Apagado	Aceso
2	Aceso	Apagado
3	Aceso	Aceso

Solução:

A finalidade deste problema é demonstrar a ação do rebote da chave. Pode ser que se observe a ação dos rebotes em algumas placas e em outras com mais dificuldade. Com os dois *leds* temos 4 estados, como mostrado na Tabela 3.7. A cada acionamento da chave S1, avançamos a contagem e comandamos os *leds*. É claro que a contagem é cíclica: 0, 1, 2, 3, 0, 1, etc. O fluxograma da Figura 3.29 resume a sub-rotina.

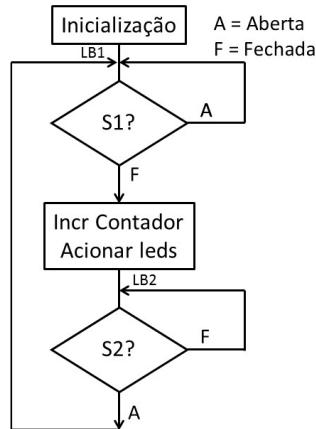


Figura 3.29. Fluxograma do ER 3.8, que constrói um contador usando LED1 e LED2, acionado pela chave S1.

Vamos construir a solução fazendo uso de um contador. Para desempenhar esse papel, poderíamos usar um registrador qualquer, porém, como a finalidade é o aprendizado, vamos reservar uma posição em memória que denominamos `CONT`. Veja que ela nada tem de especial. Apenas definimos a constante `CONT` que representa a posição de memória onde vamos construir nosso contador.

Para facilitar a preparação e o entendimento da sub-rotina, faremos uso de duas outras sub-rotinas. Uma, já conhecida, denominada `CONFIG` para configurar os pinos que são usados e outra, denominada `CONT_LEDS` que acende ou apaga os *leds* em função da contagem (posição `CONT`).

```

;ER 3.8
        CALL  #CT_SL           ;Chamar sub-rotina
        JMP   $                 ;Prender MSP
;
; Sub-rotina : inverter LED1 (P1.0) a cada o acionamento de S1 (P2.1)
;
CONT     .equ  0x2400         ;Contador na posição 0x2400
;
CT_SL:   CALL  #CONFIG       ;Configurar pinos
        MOV.B #0,&CONT          ;Contador=0
LB1:    BIT.B #BIT1,&P2IN      ;Testar S1
        JNZ   LB1              ;Se aberta, voltar
        INC.B &CONT            ;S1 Fechada: Incrementar contador

```

Ricardo Zelenovsky, Daniel Café e Eduardo Peixoto

```

        CALL  #CONT_LEDS      ;Acionar leds
LB2:   BIT.B #BIT1,&P2IN    ;Testar S1
        JZ    LB2          ;Se fechada, repetir teste
        JMP   LB1          ;Se aberta, repetir o laço
;
; Acionar os leds de acordo com o valor do contador
CONT_LEDS: BIT.B #BIT0,&CONT      ;Testar bit 0
           JZ    CL1          ;Se bit0 = 0, saltar
           BIS.B #BIT7,P4OUT    ;Bit0 = 1 --> LED2 = 0 --> aceso
           JMP   CL2          ;Próximo LED
CL1:   BIC.B #BIT7,P4OUT    ;Bit0 = 0 --> LED2 = 0 --> apagado
CL2:   BIT.B #BIT1,&CONT      ;Testar bit 1
           JZ    CL3          ;Se bit1 = 0, saltar
           BIS.B #BIT0,P1OUT    ;Bit1 = 1 --> LED1 = 1 --> aceso
           RET
CL3:   BIC.B #BIT0,P1OUT    ;Bit0 = 0 --> LED1 = 0 --> apagado
           RET
;
; Configurar os pinos
CONFIG:  BIS.B #BIT0,&P1DIR    ;Configurar P1.0 = saída
           BIC.B #BIT0,&P1OUT    ;P1.0 = 0 - LED1 apagado
           ;
           BIS.B #BIT7,&P4DIR    ;Configurar P4.7 = saída
           BIC.B #BIT7,&P4OUT    ;P4.7 = 0 - LED2 apagado
           ;
           BIC.B #BIT1,&P2DIR    ;Configurar S1 = P2.1 = entrada
           BIS.B #BIT1,&P2REN    ;Habilitar resistor
           BIS.B #BIT1,&P2OUT    ;Selecionar pullup
           RET

```

Precisamos fazer alguns comentários sobre a solução acima apresentada. O leitor pode estar preocupado com o fato de usarmos a posição `CONT` para construir um contador de 8 bits quando o problema pede um contador de apenas 2 bits. Não deveríamos fazer o contador voltar a zero quando ele ultrapassar a contagem 3? É comum esta preocupação! Mas, como é um contador binário, basta usarmos apenas os dois bits da direita e ignorarmos os demais, ou seja, usamos apenas os dois bits da direita, não importando o tamanho do contador.

A sub-rotina `CONT_LEDS` que aciona os *leds* pode ser escrita de forma mais eficiente. Veja a Figura 3.30. O fluxograma da esquerda reflete a sub-rotina recém apresentada. Em função do estado de cada *bit*, ela atualiza o respectivo *led*. A situação em que temos dois caminhos para uma decisão é mais custosa para o *assembly*, ou seja, vai implicar num maior uso de saltos. O fluxograma do lado direito é mais vantajoso. Veja que antes da decisão, colocamos o *led* no estado de apagado e só depois consultamos o estado do *bit* correspondente. Se ele estiver em zero, basta saltar adiante, caso contrário, acende-se o *led*.

O leitor poderá dizer que se força o *led* sempre a passar pelo estado de apagado e isso poderia ser ruim na situação em que o *led* já está aceso e o novo estado também é aceso. Porém, como as instruções são rápidas, provavelmente o olho do usuário não vai perceber. Podemos estimar que o *led* fica apagado apenas por 4 ciclos: apagar o *led* (1 ciclo), testar o *bit* (2 ciclos) e acender o *led* (1 ciclo). Isto vai consumir $4/1.048.576$ que é aproximadamente 3,8 μ s. Logo após a Figura 3.30 está a listagem das duas versões da sub-rotina `CONT_LED$`. Note que a versão da direita tem duas instruções a menos.

O leitor está convidado a criar uma terceira versão que trabalha ao contrário: o *led* é aceso e só depois se testa se esse estado era realmente o correto. O fato do *led* acender brevemente durante o período em que deveria ficar apagado pode ser mais perceptível. Porém, neste programa nenhuma diferença faz. Esta forma de se tomar decisão é muito proveitosa e bem eficiente quando se trata de variáveis internas de um programa (variáveis que não representam saída).

Finalmente, ao testar o programa no seu LaunchPad o leitor deve ter notado problemas na contagem. O acionamento da chave, algumas vezes, ocasionou duas contagens e não apenas uma. Este problema se deve à presença dos rebotes (*bounces*) da chave e é assunto para o próximo exercício resolvido.

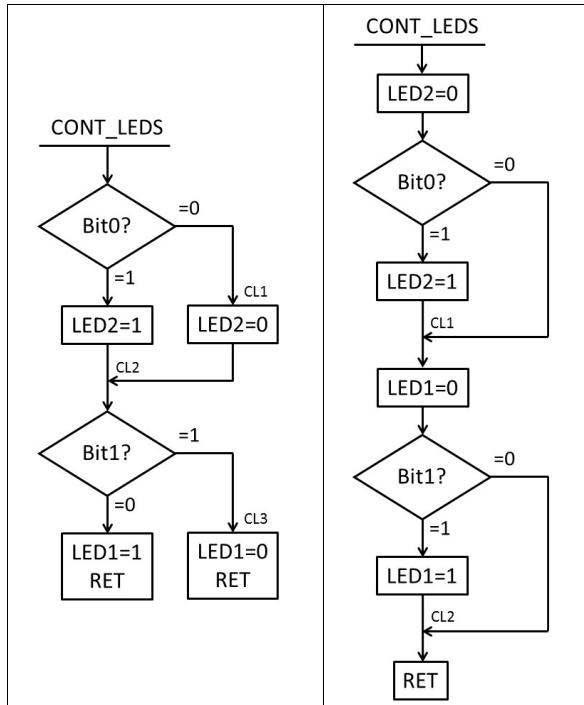


Figura 3.30. Duas propostas para a rotina que aciona os leds em função dos valores dos bits 0 e 1 do contador.

CONT_LEDS: BIT.B #BIT0, &CONT JZ CL1 BIC.B #BIT7, P4OUT JMP CL2 CL1: BIS.B #BIT7, P4OUT CL2: BIT.B #BIT1, &CONT JZ CL3 BIS.B #BIT0, P1OUT RET CL3: BIC.B #BIT0, P1OUT RET
--

CONT_LEDS: BIC.B #BIT7, P4OUT BIT.B #BIT0, &CONT JZ CL1 BIS.B #BIT7, P4OUT BIC.B #BIT0, P1OUT BIT.B #BIT1, &CONT JZ CL3 BIS.B #BIT0, P1OUT RET
--

ER 3.9. Resolva o ER 3.8, mas agora adotando uma estratégia para eliminar os rebotes da chave.

Solução:

O problema do rebote está presente em diversas situações da eletrônica. Em especial ele surge quando se muda o estado de um contato metálico, como é o caso da chave mecânica. Vimos que quando a chave está aberta, lemos 1 pelo pino corresponde e lemos 0 quando ela está fechada. O problema aparece quando se aciona ou quando se libera a chave.

A Figura 3.31 apresenta uma ilustração para esses rebotes. Note que a chave, ao ser fechada, não vai imediatamente para 0, mas sim oscila um pouco e só depois estabiliza em 0. Algo semelhante acontece quando se libera a chave. O problema é que o processador é rápido e vai interpretar esses rebotes como múltiplos acionamentos da chave, o que, no exercício anterior resultou em contagens espúrias.

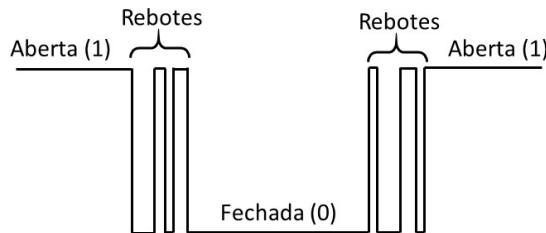


Figura 3.31. Ilustração dos rebotes por ocasião do acionamento (fechada) e liberação (aberta) de uma chave.

O intervalo de duração dos rebotes, usualmente, está abaixo de 1 ms. Assim, uma forma muito fácil de eliminá-los é “cegar” a CPU durante um breve período, logo após cada mudança de estado da chave. A Figura 3.32 apresenta um fluxograma com essa ideia. A caixa *debounce* nada mais é que uma sub-rotina que consome tempo, ou seja, durante esse tempo a CPU nada faz. A pergunta de quanto tempo deve demorar a sub-rotina de *debounce* é difícil de ser respondida. Essa demora deve ser determinada empiricamente.

A listagem a seguir é a repetição da do ER 3.8, acrescida do *debounce*. Foi então incluída a sub-rotina `DEBOUNCE`, que carrega o valor `DBC` em `R5` e depois fica decrementando `R5` até chegar a 0. Esse valor `DBC` deve ser determinado empiricamente. É preciso sugerir um valor para `DBC` e depois testar o programa contador. Caso haja rebote, aumenta-se seu valor. Ele deve ter o valor mínimo necessário, pois um `DBC` muito acima do necessário desperdiça tempo de CPU. O EP 3.8 apresenta uma outra abordagem para tratar os rebotes de chaves.

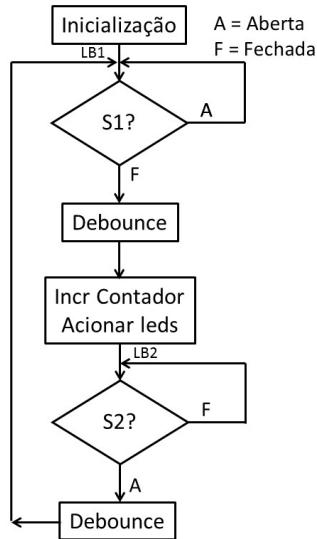


Figura 3.32. Fluxograma para se construir um contador usando LED1 e LED2, acionados pela chave S1, com tratamento de rebotes (bounces).

Listagem da solução do ER 3.9

```

;ER 3.9
        CALL  #CT_SL           ;Chamar sub-rotina
        JMP   $                 ;Prender MSP
;
; Sub-rotina : inverter LED1 (P1.0) a cada o acionamento de S1 (P2.1)
;
CONT     .equ   0x2400          ;Contador na posição 0x2400
DBC      .equ   10000          ;Chute para o debounce
;
CT_SL:   CALL   #CONFIG         ;Configurar pinos
        MOV.B #0,&CONT          ;Contador=0
LB1:     BIT.B #BIT1,&P2IN       ;Testar S1
        JNZ   LB1              ;Se aberta, voltar
        CALL  #DEBOUNCE         ;Realizar debounce
        INC.B &CONT            ;S1 Fechada: Incrementar contador
        CALL  #CONT_LEDS        ;Acionar leds
LB2:     BIT.B #BIT1,&P2IN       ;Testar S1
        JZ    LB2              ;Se fechada, repetir teste
        CALL  #DEBOUNCE         ;Realizar debounce
        JMP   LB1              ;Se aberta, repetir o laço
;
; Acionar os leds de acordo com o valor do contador
CONT_LEDS: BIT.B #BIT0,&CONT      ;Testar bit 0
  
```

```

JZ      CL1           ;Se bit0 = 0, saltar
BIS.B #BIT7,P4OUT    ;Bit0 = 1 --> LED2 = 0 --> aceso
JMP    CL2           ;Próximo LED
CL1:   BIC.B #BIT7,P4OUT    ;Bit0 = 0 --> LED2 = 0 --> apagado
CL2:   BIT.B #BIT1,&CONT    ;Testar bit 1
      JZ    CL3           ;Se bit1 = 0, saltar
      BIS.B #BIT0,P1OUT    ;Bit1 = 1 --> LED1 = 1 --> aceso
      RET
CL3:   BIC.B #BIT0,P1OUT    ;Bit0 = 0 --> LED1 = 0 --> apagado
      RET
;
; Rotina para eliminar os rebotes (bounces)
DEBOUNCE: MOV   #DBC,R5        ;Inicializar contador (R5)
DB1:    DEC   R5          ;Decrementar contador
        JNZ  DB1         ;se <> 0, repetir
        RET
;
; Configurar os pinos
CONFIG:  BIS.B #BIT0,&P1DIR    ;Configurar P1.0 = saída
        BIC.B #BIT0,&P1OUT    ;P1.0 = 0 - LED1 apagado
;
        BIS.B #BIT7,&P4DIR    ;Configurar P4.7 = saída
        BIC.B #BIT7,&P4OUT    ;P4.7 = 0 - LED2 apagado
;
        BIC.B #BIT1,&P2DIR    ;Configurar S1 = P2.1 = entrada
        BIS.B #BIT1,&P2REN     ;Habilitar resistor
        BIS.B #BIT1,&P2OUT     ;Selecionar pullup
        RET

```

ER 3.10. Repita o ER 3.8, mas agora sem usar o *pullup* interno, ou seja, deixando a entrada em alta impedância e sem tratar os rebotes.

Solução:

Este exercício tem o objetivo de ilustrar o perigo que representa deixar, sem necessidade, uma entrada em alta-impedância. A chave S1 ainda vai comandar o contador. Entretanto, podem acontecer acionamentos indevidos enquanto ela estiver aberta. Toque por baixo da placa, na posição onde está a chave S1 e o leitor notará acionamentos indevidos. Veja que a listagem é idêntica à do ER 3.8, à exceção das duas últimas linhas.

Pergunta: mudaria alguma coisa no comportamento do contador binário se incluíssemos a sub-rotina de *debounce* desenvolvida no exercício anterior?

Listagem da solução do ER 3.10

```

;ER 3.10
CALL  #CT_SL          ;Chamar sub-rotina
JMP   $               ;Prender MSP

```

Ricardo Zelenovsky, Daniel Café e Eduardo Peixoto

```

;
; Sub-rotina : inverter LED1 (P1.0) a cada o acionamento de S1 (P2.1)
;
CONT      .equ   0x2400          ;Contador na posição 0x2400
;
CT_S1:    CALL   #CONFIG        ;Configurar pinos
          MOV.B #0,&CONT      ;Contador=0
LB1:      BIT.B #BIT1,&P2IN     ;Testar S1
          JNZ   LB1           ;Se aberta, voltar
          INC.B &CONT        ;S1 Fechada: Incrementar contador
          CALL   #CONT_LED
LB2:      BIT.B #BIT1,&P2IN     ;Testar S1
          JZ    LB2           ;Se fechada, repetir teste
          JMP   LB1           ;Se aberta, repetir o laço
;
; Acionar os leds de acordo com o valor do contador
CONT_LED: BIT.B #BIT0,&CONT      ;Testar bit 0
          JZ    CL1           ;Se bit0 = 0, saltar
          BIS.B #BIT7,P4OUT    ;Bit0 = 1 --> LED2 = 0 --> aceso
          JMP   CL2           ;Próximo LED
CL1:      BIC.B #BIT7,P4OUT    ;Bit0 = 0 --> LED2 = 0 --> apagado
CL2:      BIT.B #BIT1,&CONT      ;Testar bit 1
          JZ    CL3           ;Se bit1 = 0, saltar
          BIS.B #BIT0,P1OUT    ;Bit1 = 1 --> LED1 = 1 --> aceso
          RET
CL3:      BIC.B #BIT0,P1OUT    ;Bit0 = 0 --> LED1 = 0 --> apagado
          RET
;
; Configurar os pinos
CONFIG:   BIS.B #BIT0,&P1DIR    ;Configurar P1.0 = saída
          BIC.B #BIT0,&P1OUT    ;P1.0 = 0 - LED1 apagado
          ;
          BIS.B #BIT7,&P4DIR    ;Configurar P4.7 = saída
          BIC.B #BIT7,&P4OUT    ;P4.7 = 0 - LED2 apagado
          ;
          BIC.B #BIT1,&P2DIR    ;Configurar S1 = P2.1 = entrada
          BIC.B #BIT1,&P2REN    ;Desabilitar resistor
          RET

```

ER 3.11. Este exercício amplia a operação do contador proposto no ER3.9. Escreva a sub-rotina CT_S1 que implementa um contador binário com os dois *leds* da placa. A cada acionamento de S1 (transição de aberta para fechada), o contador avança uma contagem e a cada acionamento de S2 (transição de aberta para fechada), o contador recua uma contagem. Em outras palavras:

- S1 → incrementa o contador e
- S2 → decrementa o contador.

Solução:

Até agora, fizemos o monitoramento da chave prendendo a execução do processador. Em outras palavras, a CPU ficava parada esperando o acionamento ou a liberação da chave. A situação agora é diferente, pois vamos precisar monitorar ao mesmo tempo o estado das duas chaves. Neste caso, para saber se a chave passou do estado aberta para o estado fechada, será necessário armazenar o último estado de cada chave para compará-lo com o atual.

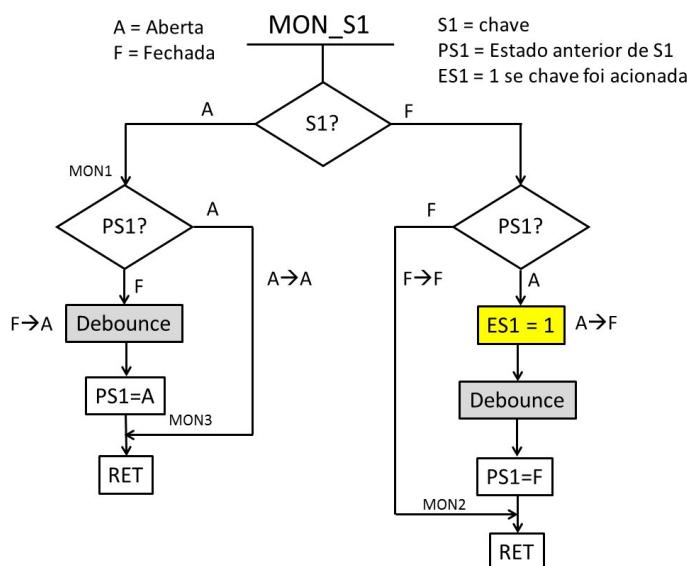


Figura 3.33. Fluxograma de uma sub-rotina para monitorar a chave S1, sem prender a execução. Note que no caso deste fluxograma a ação acontece quando se aciona da chave.

O fluxograma da Figura 3.33 apresenta a proposta de uma sub-rotina para monitorar a chave S1, sem prender a execução. Será necessário armazenar o estado anterior da chave para compará-lo com o atual e então determinar se ela foi acionada. A variável PS1 guarda o estado anterior da chave (passado de S1). Assim, lemos o estado atual da chave e o comparamos com estado anterior (PS1), conforme detalhado a seguir:

$S1 \rightarrow$ Estado atual da chave
 $S1 = A \rightarrow$ chave S1 está aberta
 $S1 = F \rightarrow$ chave S1 está fechada

$PS1 \rightarrow$ estado anterior da chave
 $PS1 = A \rightarrow$ chave S1 estava aberta
 $PS1 = F \rightarrow$ chave S1 estava fechada

Então teremos as 4 possibilidades listadas abaixo e indicadas no fluxograma.

- PS1 = A e S1 = A: A → A chave estava aberta e continua aberta;
- PS1 = F e S1 = A: F → A chave estava fechada e foi liberada;
- PS1 = A e S1 = F: A → F chave estava aberta e foi acionada e
- PS1 = F e S1 = F: F → F chave estava fechada e continua fechada;

Abaixo está o detalhamento da sub-rotina MON_S1.

Sub-rotina MON_S1:

Recebe: Nada;

Retorna: ES1 = 1 se a chave passou de aberta para fechada.

Recursos a serem usados pela sub-rotina:

PS1 = estado anterior da chave (passado da chave 1);

Usa sub-rotina DEBOUNCE.

A listagem abaixo apresenta a solução para a sub-rotina que monitora a chave S1. Uma sub-rotina semelhante deverá ser preparada para a chave S2.

```
; Sub-rotina para monitorar S1 (P2.1)
; Retorna ES1 = 1 se a chave passou de aberta para fechada
MON_S1:    BIT.B #BIT1,&P2IN      ;Testar S1
            JNZ  MON1        ;Se aberta, saltar
            CMP  #FECHADA,&PS1   ;Passado = Fechada ?
            JZ   MON2        ;Se fechada, seguir adiante
            MOV  #1,&ES1       ;ES1 = 1, marcar chave acionada
            CALL #DEBOUNCE     ;Fazer debounce
            MOV  #FECHADA,&PS1   ;Estado anterior = fechada
MON2:      RET             ;Retornar
;
MON1:      CMP  #ABERTA,&PS1    ;Passado = Aberta ?
            JZ   MON3        ;Se aberta, seguir adiante
            CALL #DEBOUNCE     ;Fazer debounce
            MOV  #ABERTA,&PS1    ;Estado anterior = aberta
MON3:      RET             ;Retornar
```

A sub-rotina acima faz uso de constantes que devem ser definidas pelo programa chamador. Note que o uso de constantes com nomes ABERTA e FECHADA facilita muito o entendimento da sub-rotina.

Um ponto muito importante a comentar é que essa sub-rotina apenas faz ES1 = 1 quando a chave passa de aberta para fechada. O programa chamador, após perceber o acionamento da chave S1, tem a obrigação de fazer ES1 = 0. Muito leitores poderiam ficar tentados a deixar a sub-rotina MON_S1 fazer ES1 = 0 só quando a chave passar de fechada para aberta. Isso é ruim porque ES1 passa a acompanhar a velocidade da chave,

que é lenta comparada com o processador. Ou seja, se fizer isso, o contador fica contado repetitivamente até a liberação da chave.

A Figura 3.34 apresenta a solução para o problema proposto. É de se notar que com as sub-rotinas MON_S1 e MON_S2 ela ficou muito simples. A listagem correspondente está apresentada logo em seguida. Veja que a estruturação em sub-rotinas em muito facilita a escrituração e compreensão do programa.

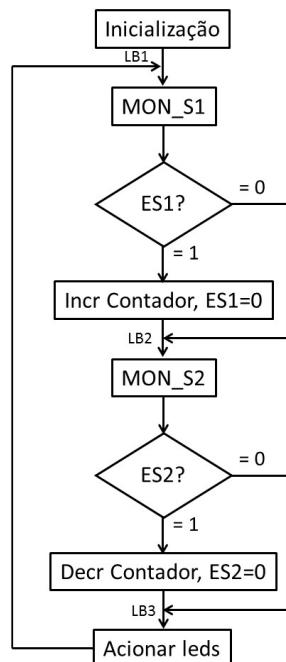


Figura 3.34. Fluxograma para o controle de um contador binário usando as chaves S1 e S2.

Listagem da solução do ER 3.11

```

;ER 3.11
CALL  #CT_2SL           ;Chamar sub-rotina
JMP   $                 ;Prender MSP
;
; Sub-rotina : Contador de 2 bits, S1 incrementa e S2 decrementa
;
; Constantes
FECHADA .equ 0          ;Zero representa chave fechada

```

```

ABERTA    .equ 1          ;Um representa chave aberta
DBC      .equ 10000       ;Chute para o debounce
; Variáveis
VAR      .equ 0x2400      ;Área de variáveis, todas de 16 bits
PS1      .equ VAR         ;Passado da chave S1
PS2      .equ VAR+2       ;Passado da chave S2
ES1      .equ VAR+4       ;vai para 1 qdo S1 vai de A-->F
ES2      .equ VAR+6       ;vai para 1 qdo S2 vai de A-->F
CONT     .equ VAR+8       ;Contador binário
;
CT_2SL:  CALL #CONFIG      ;Configurar pinos
          MOV #ABERTA,&PS1      ;Iniciar estado anterior = aberta
          MOV #ABERTA,&PS2      ;Iniciar estado anterior = aberta
          MOV #0,&CONT        ;Contador=0
LB1:     CALL #MON_S1      ;Monitorar S1
          TST &ES1           ;ES1 = 0?
          JZ  LB2             ;Salta se S1 não acionada
          INC &CONT          ;S1 acionada: Incrementar contador
          CLR &ES1           ;ES1=0, já respondeu à chave
          ;
LB2:     CALL #MON_S2      ;Monitorar S2
          TST &ES2           ;ES2 = 0?
          JZ  LB3             ;Salta se S2 não acionada
          DEC &CONT          ;S2 acionada: Decrementar contador
          CLR &ES2           ;ES2=0, já respondeu à chave
LB3:     CALL #CONT_LEDS    ;Atualizar Leds
          JMP LB1             ;Repetir o laço
;
; Acionar os leds de acordo com o valor do contador
CONT_LEDS: BIT  #BIT0,&CONT      ;Testar bit 0
            JZ  CL1             ;Se bit0 = 0, saltar
            BIS.B #BIT7,P4OUT    ;Bit0 = 1 --> LED2 = 0 --> aceso
            JMP CL2             ;Próximo LED
CL1:     BIC.B #BIT7,P4OUT    ;Bit0 = 0 --> LED2 = 0 --> apagado
CL2:     BIT  #BIT1,&CONT      ;Testar bit 1
            JZ  CL3             ;Se bit1 = 0, saltar
            BIS.B #BIT0,P1OUT    ;Bit1 = 1 --> LED1 = 1 --> aceso
            RET
CL3:     BIC.B #BIT0,P1OUT    ;Bit0 = 0 --> LED1 = 0 --> apagado
            RET
;
; Sub-rotina para monitorar S1 (P2.1)
; Retorna ES1 = 1 se a chave passou de aberta para fechada
MON_S1:  BIT.B #BIT1,&P2IN      ;Testar S1
            JNZ MON1            ;Se aberta, saltar
            CMP #FECHADA,&PS1     ;Passado = Fechada ?
            JZ  MON2             ;Se fechada, seguir adiante
            MOV #1,&ES1           ;ES1 = 1, marcar chave acionada
            CALL #DEBOUNCE        ;Fazer debounce
            MOV #FECHADA,&PS1     ;Estado anterior = fechada

```

```

MON2:      RET           ;Retornar
;
MON1:      CMP  #ABERTA,&PS1    ;Passado = Aberta ?
      JZ   MON3          ;Se aberta, seguir adiante
      CALL #DEBOUNCE     ;Fazer debounce
      MOV   #ABERTA,&PS1    ;Estado anterior = aberta
MON3:      RET           ;Retornar
;
; Sub-rotina para monitorar S2 (P1.1)
; Retorna ES2 = 1 se a chave passou de aberta para fechada
MON_S2:    BIT.B #BIT1,&P1IN    ;Testar S2
      JNZ  MON4          ;Se aberta, saltar
      CMP  #FECHADA,&PS2    ;Passado = Fechada ?
      JZ   MON5          ;Se fechada, seguir adiante
      MOV   #1,&ES2        ;ES2 = 1, marcar chave acionada
      CALL #DEBOUNCE     ;Fazer debounce
      MOV   #FECHADA,&PS2    ;Estado anterior = fechada
MON5:      RET           ;Retornar
;
MON4:      CMP  #ABERTA,&PS2    ;Passado = Aberta ?
      JZ   MON6          ;Se aberta, seguir adiante
      CALL #DEBOUNCE     ;Fazer debounce
      MOV   #ABERTA,&PS2    ;Estado anterior = aberta
MON6:      RET           ;Retornar
;
; Rotina para eliminar os rebotes (bounces)
DEBOUNCE:  MOV   #DBC,R5      ;Inicializar contador (R5)
DB1:       DEC   R5          ;Decrementar contador
      JNZ  DB1           ;se <> 0, repetir
      RET           ;senão, retornar
;
; Configurar os pinos: LED1, LED2, S1, S2
CONFIG:    BIS.B #BIT0,&P1DIR    ;Configurar P1.0 = saída
      BIC.B #BIT0,&P1OUT     ;P1.0 = 0 - LED1 apagado
      ;
      BIS.B #BIT7,&P4DIR    ;Configurar P4.7 = saída
      BIC.B #BIT7,&P4OUT     ;P4.7 = 0 - LED2 apagado
      ;
      BIC.B #BIT1,&P2DIR    ;Configurar S1 = P2.1 = entrada
      BIS.B #BIT1,&P2REN     ;Habilitar resistor
      BIS.B #BIT1,&P2OUT     ;Selecionar pullup
      ;
      BIC.B #BIT1,&P1DIR    ;Configurar S2 = P1.1 = entrada
      BIS.B #BIT1,&P1REN     ;Habilitar resistor
      BIS.B #BIT1,&P1OUT     ;Selecionar pullup
      RET

```

Para finalizar, vamos propor dois desafios:

Ricardo Zelenovsky, Daniel Café e Eduardo Peixoto

Desafio 1) As variáveis PS1, PS2, ES1, ES2 e CONT são de 16 bits. Isso é um pequeno desperdício de memória. Modifique o programa de forma a que essas variáveis sejam de 8 bits.

Desafio 2) O leitor poderá verificar que as chaves trabalham de forma independente. A decisão tomada com a chave S1 não verifica se a chave S2 está aberta ou fechada. Uma proposta interessante seria permitir que somente uma chave estivesse acionada por vez. Será que o leitor consegue escrever uma solução para este problema?

ER 3.12. Este exercício pede para escrever a sub-rotina JOGO, lembra uma disputa entre dois jogadores onde vence quem primeiro acionar sua chave, S1 ou S2. O jogo inicia com os dois *leds* apagados, quando então os dois jogadores podem agir sobre suas chaves. Ganha aquele que acionar primeiro sua chave. Se for a chave S1, o LED1 acende. Se for a chave S2, o LED2 acende. O jogo só recomeça quando as duas chaves estiverem soltas e os leds apagados.

Observação: O leitor pode experimentar uma certa insegurança para caracterizar o início da disputa. O EP 3.12 propõe uma melhora. Por enquanto vamos imaginar que existe um juiz que, contando “1, 2, 3 e já”, dá início à disputa.

Solução:

Novamente precisamos monitorar as duas chaves, sem prender a o processador. O interessante é que neste exercício não precisamos de nos preocupar com os rebotes (será que realmente não precisamos nos preocupar com os rebotes?). A solução é simples e está esquematizada no fluxograma da Figura 3.35. O laço começa apagando os *leds* e depois, usando dois testes, faz o desvio quando uma das duas chaves estiver acionada, acendendo o *led* correspondente. Depois, o programa fica preso em um outro laço, enquanto uma das chaves estiver acionada.

A solução apresentada tem uma deficiência. Se o processador for energizado com uma tecla já acionada, esta tecla será a vencedora. O ideal seria que o jogo só começasse com as duas teclas liberadas. Analisando a Figura 3.35, poderíamos pensar em trocar de posição os laços que monitoram as chaves. Fica a cargo do leitor fazer esta alteração e testá-la.

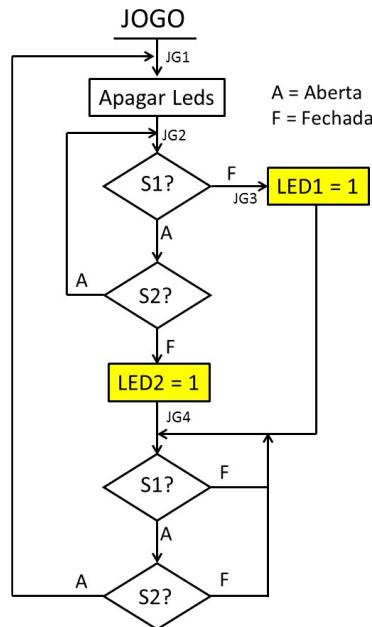


Figura 3.35. Fluxograma para a disputa de quem aciona primeiro sua chave.

Listagem da solução do ER 3.12

```

;ER 3.12
        CALL  #JOGO           ;Chamar sub-rotina
        JMP   $               ;Prender MSP
;
; Sub-rotina : Disputa para ver quem primeiro aciona sua chave
JOGO:   CALL  #CONFIG        ;Configurar pinos
        BIC.B #BIT0,P1OUT      ;Apagar LED1
        BIC.B #BIT7,P4OUT      ;Apagar LED2
JG1:    BIT.B #BIT1,P1IN      ;Testar S1 (P2.1)
        JZ    JG3              ;S1 Ganhou
        BIT.B #BIT1,P1IN      ;Testar S2 (P1.1)
        JNZ   JG2              ;S1 e S2 abertas, repetir laço
        BIS.B #BIT7,P4OUT      ;S2 ganhou, LED2 aceso
        JMP   JG4              ;Saltar para esperar soltar chaves
JG3:    BIS.B #BIT0,P1OUT      ;LED1 aceso
;
JG4:    BIT.B #BIT1,P2IN      ;Testar S1 (P2.1)
        JZ    JG4              ;S1 fechada, repetir laço
        BIT.B #BIT1,P2IN      ;Testar S2 (P1.1)
        JZ    JG4              ;S2 fechada, repetir laço

```

```

        JMP    JG1           ;S1 e S2 abertas, reiniciar jogo
;
; Configurar os pinos: LED1, LED2, S1, S2
CONFIG:   BIS.B #BIT0,&P1DIR      ;Configurar P1.0 = saída
          BIC.B #BIT0,&P1OUT      ;P1.0 = 0 - LED1 apagado
          ;
          BIS.B #BIT7,&P4DIR      ;Configurar P4.7 = saída
          BIC.B #BIT7,&P4OUT      ;P4.7 = 0 - LED2 apagado
          ;
          BIC.B #BIT1,&P2DIR      ;Configurar S1 = P2.1 = entrada
          BIS.B #BIT1,&P2REN      ;Habilitar resistor
          BIS.B #BIT1,&P2OUT      ;Selecionar pullup
          ;
          BIC.B #BIT1,&P1DIR      ;Configurar S2 = P1.1 = entrada
          BIS.B #BIT1,&P1REN      ;Habilitar resistor
          BIS.B #BIT1,&P1OUT      ;Selecionar pullup
          RET

```

ER 3.13. Este exercício é um pouco longo, porém trabalha conceitos que serão importantes quando estudarmos os *timers*. Aqui, pretendemos controlar o brilho do LED1 (vermelho, em P1.0). O olho humano não consegue perceber variações de luminosidade quando um *led* pisca numa frequência acima de 30 Hz. Vamos escolher uma frequência bem mais alta. No caso selecionamos 50 Hz para facilitar as contas. Se fizermos o LED1 piscar na frequência de 50 Hz (período de 20 ms), ele fica aceso por 10 ms e apagado por 10 ms, ou seja, 50% do tempo aceso e 50% do tempo apagado. Se, mantendo o período de 20 ms, alterarmos o tempo em que ele fica aceso ou apagado, poderemos controlar seu brilho. Isto é denominado de Modulação por Largura de Pulso, PWM (do inglês *Pulse Width Modulation*). A Figura 3.36 apresenta 4 exemplos de valores de PWM. O sinal em nível alto indica LED1 aceso e em nível baixo, LED1 apagado. Não foi representado, mas 0% corresponde ao LED1 sempre apagado

Esclarecido este conceito, o presente exercício pede a sub-rotina PWM1, que inicia com PWM = 0 e obedece aos comandos abaixo:

S1 → aumenta a potência em passos de 25% e

S2 → diminui a potência em passos de 25%.

Note que o máximo é 100% e o mínimo é 0%.

Observação: a percepção de brilho pelo olho humano não é linear e pode ser que o leitor pouca diferença perceba quando o brilho variar entre 75% e 100%.

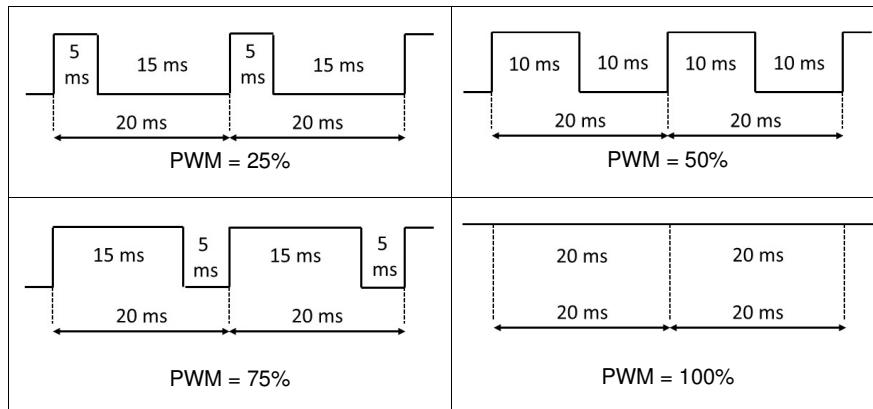


Figura 3.36. Quatro condições para o PWM (valores de ciclo de carga), sendo que nível alto indica LED1 aceso e nível baixo, LED1 apagado.

Solução:

Analisando a Figura 3.36, fica claro que vamos precisar de um atraso de 5 ms. No ER 3.1 criamos uma sub-rotina `TEMPO` (repetida abaixo) que quando foi chamada usando `TP = 65.535` resultou num atraso de 185,6 ms. Como precisamos de um atraso de 5 ms, o valor de `TP` deve ser alterado para 1.766, de acordo com a conta abaixo:

$$TP = \frac{5}{185,6} \cdot 65535 = 1766$$

```
; Sub-rotina para consumir tempo
TEMPO:    MOV    #TP,R5           ;Inicializar contador
TEMP1:    DEC    R5           ;Decrementar contador
          JNZ    TEMP1         ;Retornar quando chegar a zero
          RET
```

Já sabemos como gerar o atraso de 5 ms, resta o problema de controlar o tempo em que o LED1 fica aceso ou apagado de acordo com o PWM desejado. O leitor pode imaginar várias maneiras de fazer isso, porém vamos propor uma solução bem estruturada, que está apresentada na Figura 3.37 e cujo fluxograma está na Figura 3.38, junto com sua explicação.

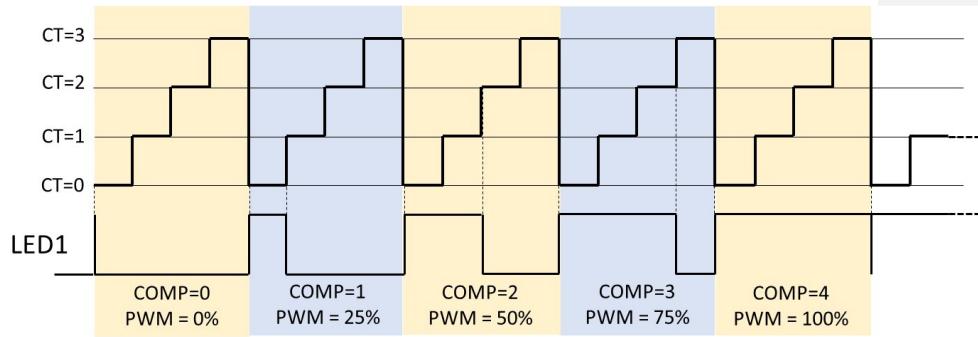


Figura 3.37. Gráfico mostrando o comportamento do Contador de Tempo (denominado de CT) e do Comparador (denominado de COMP) para gerar diversas cargas de PWM.

Na Figura 3.37, os degraus representam a contagem de tempo feita por CT (Contador de Tempo) e, na parte inferior, está sinalizado com comportamento do LED1, de acordo com o valor de COMP (comparador). São apresentadas as 5 possibilidades (0%, 25%, 50%, 75%, 100%) pedidas, selecionadas pelo valor de COMP. Pode parecer estranho que quando COMP = 0, surja um pequeno pulso na saída, ou seja, o LED1 dá uma piscada rápida. Isso acontece pela estrutura do programa. Explicando melhor: quando CT = 0, o LED1 é aceso e, na comparação que é feita em seguida, se constata que CT = COMP e o LED1 é apagado, vide Figura 3.38.

Desafio 1: o leitor consegue alterar o programa de forma a eliminar esta pequena piscada do LED1 quando PWM = 0%?

A listagem solução é apresentada em seguida. Nessa listagem, para evitar que ela ocupasse muito espaço, foram omitidas das sub-rotinas `MON_S1`, `MON_S2` e `DEBOUNCE`. Essas sub-rotinas estão disponíveis na solução do E.R. 3.11. Note que, para que o leitor experimente outros períodos, deixamos comentadas as opções para que a sub-rotina `TEMPO` opere com os atrasos de:

- `TP10MS` → 10 ms, o que corresponde a 25 Hz (4 x 10 ms)
- `TP5MS` → 5 ms, o que corresponde a 100 Hz (4 x 5 ms)
- `TP2P5MS` → 2,5 ms, o que corresponde a 200 Hz (4 x 2,5 ms)

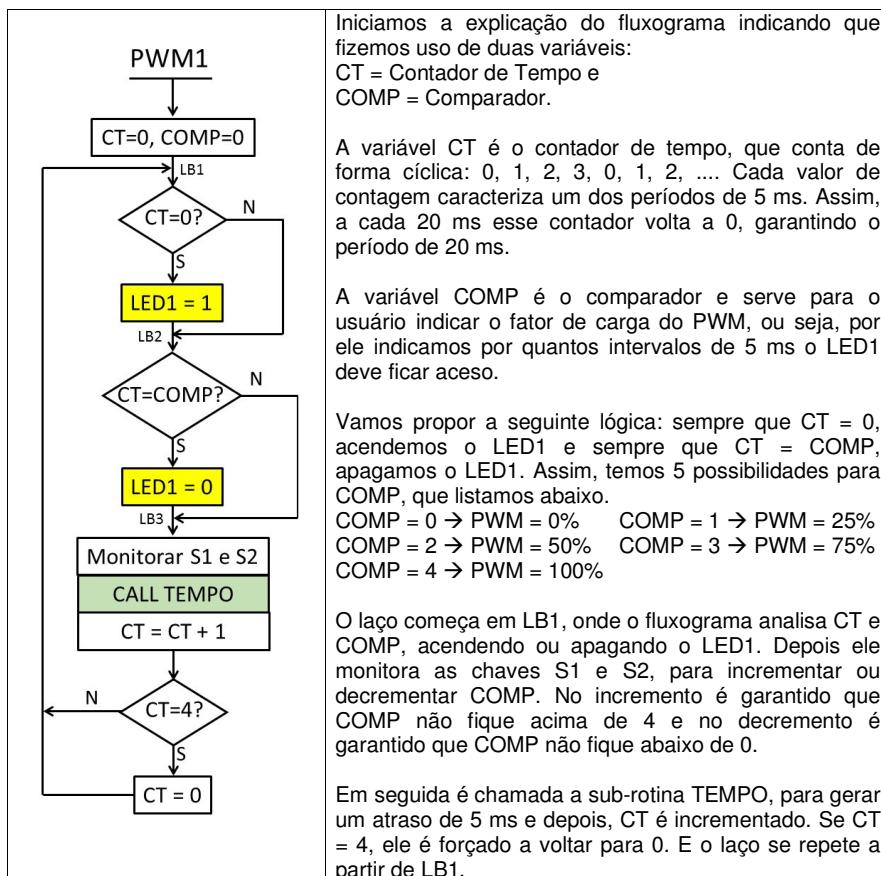


Figura 3.38. Fluxograma para realizar PWM com sua explicação ao lado. Este fluxograma deve ser analisado com o diagrama de tempo da Figura 3.37.

Listagem da solução do ER 3.13

```
;ER 3.13
CALL #PWM1           ;Chamar sub-rotina
JMP    $              ;Prender MSP
;
; PWM com 5 passos: 0%, 25%, 50%, 75% e 100%
; S1 = P2.1 e S2 = P1.1
;
; Constantes:
TP10MS    .equ 3532      ;Atraso de 10 ms
TP5MS     .equ 1766      ;Atraso de 5 ms
```

```

TP2P5MS .equ 883      ;Atraso de 2,5 ms
FECHADA .equ 0        ;Zero representa chave fechada
ABERTA   .equ 1        ;Um representa chave aberta
DBC      .equ 10000    ;Chute para o debounce
;DBC     .equ 1        ;Debounce muito curto
;
; Variáveis:
VAR      .equ 0x2400  ;Área de variáveis
CT       .equ VAR      ;Contador de tempo (0, 1, 2, 3, 4)
COMP    .equ VAR+2    ;Comparador
PS1     .equ VAR+4    ;Passado da chave S1
PS2     .equ VAR+6    ;Passado da chave S2
ES1     .equ VAR+8    ;vai para 1 qdo S1 vai de A-->F
ES2     .equ VAR+10   ;vai para 1 qdo S2 vai de A-->F
;
; Início da Sub-rotina
PWM1:   CALL  #CONFIG  ;Configurar pinos
        CLR   &CT      ;Zerar CT = Contador de Tempo
        CLR   &COMP    ;Zerar COMP = Comparador
LB1:    TST   &CT      ;CT = 0
        JNZ   LB2      ;Se CT <> 0 saltar
        BIS.B #BIT0,P1OUT ;Se CT = 0 --> Acender LED1
LB2:    CMP   &COMP,&CT  ;CT = COMP ?
        JNZ   LB3      ;Se CT <> COMP, saltar
        BIC.B #BIT0,P1OUT ;Se CT = COMP --> Apagar LED1
        ;
LB3:   CALL  #TEMPO    ;Chamar atraso
        CALL  #MON_S1   ;Monitorar S1
        CALL  #MON_S2   ;Monitorar S2
        ;
        TST   &ES1     ;S1 acionada?
        JZ    LB4      ;Não, saltar
        CLR   &ES1     ;Sim, indicador ES1 = 0
        CMP   #4,&COMP  ;COMP = 4 (limite superior)?
        JZ    LB4      ;Caso positivo, saltar
        INC   &COMP    ;Caso negativo, incrementar
        ;
LB4:   TST   &ES2     ;S2 acionada?
        JZ    LB5      ;Não, saltar
        CLR   &ES2     ;sim, indicador ES2 = 0
        TST   &COMP    ;COMP = 0 (limite inferior)
        JZ    LB5      ;Caso positivo, saltar
        DEC   &COMP    ;Caso negativo, decrementar
        ;
LB5:   INC   &CT      ;Incrementar Contador de Tempo
        CMP   #4,&CT    ;CT ultrapassou 3?
        JNZ   LB1      ;Caso negativo, voltar ao laço
        CLR   &CT      ;Caso positivo, CT = 0
        JMP   LB1      ;Voltar ao laço
;

```

```

; Colocar aqui a sub-rotina MON_S1 do ER 3.11
; Colocar aqui a sub-rotina MON_S2 do ER 3.11
; Colocar aqui a sub-rotina DEBOUNCE do ER 3.11
;
; Sub-rotina para consumir tempo
TEMPO:
    MOV #TP10MS,R5 ; Inicializar contador, 10 ms
    MOV #TP5MS,R5 ; Inicializar contador, 5 ms
    MOV #TP2P5MS,R5 ; Inicializar contador, 2,5 ms
TEMP1: DEC R5 ; Decrementar contador
    JNZ TEMP1 ; Retornar quando chegar a zero
    RET
;
; Configurar os pinos: LED1, LED2, S1, S2
CONFIG: BIS.B #BIT0,&P1DIR ; Configurar P1.0 = saída
        BIC.B #BIT0,&P1OUT ; P1.0 = 0 - LED1 apagado
        ;
        BIC.B #BIT1,&P2DIR ; Configurar S1 = P2.1 = entrada
        BIS.B #BIT1,&P2REN ; Habilitar resistor
        BIS.B #BIT1,&P2OUT ; Selecionar pullup
        ;
        BIC.B #BIT1,&P1DIR ; Configurar S2 = P1.1 = entrada
        BIS.B #BIT1,&P1REN ; Habilitar resistor
        BIS.B #BIT1,&P1OUT ; Selecionar pullup
        RET

```

Ao provar o programa proposto, o leitor deve notar que, ao acionar uma das chaves, algumas vezes o LED1 tem um comportamento estranho. Isso acontece por duas razões.

A primeira razão é a presença da sub-rotina `DEBOUNCE`, que introduz um longo atraso no laço de programa que deveria demorar apenas 5 ms. Veja que a sub-rotina `TEMPO` usa 1.766 (constante `TP5MS`) repetições e a sub-rotina `DEBOUNCE` usa 1.000 (constante `DBC`) repetições. Assim, cada vez que uma tecla é acionada ou liberada, a sub-rotina `DEBOUNCE` aumenta o laço em 56%. Tente repetir o programa, mas usando `DBC` igual a 1 (ficamos vulneráveis aos rebotes). Você vai notar que o comportamento do LED1 melhora muito. Porém, ao acionar várias vezes as duas chaves, vai ainda notar que algumas vezes a chave `S2` provoca uma piscada mais forte. Por que será?

Esta é a segunda razão e pede uma explicação mais cuidadosa. O acionamento da chave e a correspondente alteração de `COMP` ocorrem de forma aleatória em relação ao contador `CT`. Não se sabe se o valor de `COMP` será alterado com `CT = 0` ou com `CT = 1`, etc. Para explicar melhor esse problema, vamos usar o diagrama de tempo apresentado na Figura 3.39. Nesta figura, para facilitar a explicação, consideramos que o contador `CT` vai de 0 até o valor `MAX` e volta a 0. No caso do programa, tínhamos `MAX = 3`. As rampas representam o Contador de Tempo (`CT`) progredindo na contagem. O comparador `COMP` pode ter qualquer valor dentro desta faixa.

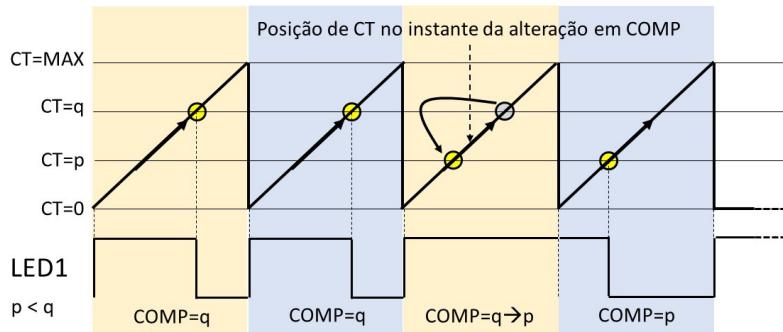


Figura 3.39. Diagrama de tempo mostrando possível problema na saída LED1 causa pela alteração de COMP num momento inconveniente.

Na figura, no terceiro período, COMP tinha o valor q e foi alterado para p , porém essa alteração ocorreu quando o contador CT estava num valor entre p e q . Neste período não vai acontecer $CT = COMP$, pois COMP foi alterado para um valor abaixo do valor atual de CT e por isso o LED1 não é desligado. Surge então uma piscada forte pois o led permanece aceso durante todo o período.

Para resolver esse problema podemos propor a seguinte alteração. As chaves S1 e S2 não alteram diretamente a variável COMP, mas sim uma variável auxiliar que vamos chamar de COMP_AUX. Toda vez que CT voltar para zero, copia-se em COMP o valor de COMP_AUX. Isto recebe o nome de dupla *bufferização*. Assim, em toda rampa, em algum lugar teremos $CT = COMP$.

Desafio 2: use o recurso de dupla *bufferização*, recém explicada, para evitar a piscadela do LED1 quando uma das chaves é acionada. Veja EP 3.16.

Para finalizar este exercício, vamos comentar sobre a precisão no tempo. Será que estamos trabalhando exatamente com intervalos de 5 ms? Será que o período do PWM é de 20 ms? É claro que não! A sub-rotina TEMPO gasta 5 ms, mas existem muitas outras instruções que são executadas entre cada chamada desta sub-rotina. Quando acionamos alguma chave, soma-se ainda o tempo da sub-rotina DEBOUNCE. Na verdade, devemos estar com um período acima de 20 ms.

No presente caso, este erro é tolerável pois a quantidade extra de instruções é pequena se comparada com as 1.766 repetições da sub-rotina TEMPO. Porém, se precisarmos diminuir o período do PWM, a proporção do erro temporal irá aumentar. Tudo isso será muito bem resolvido quando estudarmos os *timers*.

ER 3.14. Um *led* bicolor traz dois *leds* de cores diferentes em um único encapsulamento. Ele é interessante porque possibilita uma sinalização diferente do usual e permite provar a combinação de duas cores. No presente caso vamos usar um *led* bicolor verde e vermelho. Nosso olho interpreta a combinação dessas duas cores como amarelo.

Vamos ao exercício. Escreva a sub-rotina BI_LED que aciona um *led* bicolor (verde e vermelho) que está conectado aos pinos P2.4 e P2.5, como mostrado na Figura 3.40. Esta sub-rotina deve fazer o *led* piscar na frequência de 100 Hz (5 ms aceso e 5 ms apagado), seguindo os modos listados abaixo. A cada acionamento de S1 ou de S2, a sub-rotina avança para o modo seguinte de forma cíclica: 1, 2, 3, 4, 1, 2, ...

- Modo 1: Piscar o *led* verde na frequência de 100 Hz;
- Modo 2: Piscar o *led* vermelho na frequência de 100 Hz;
- Modo 3: Piscar sincronamente os *leds* verde e vermelho na frequência de 100 Hz;
- Modo 4: Piscar alternadamente os *leds* verde e vermelho na frequência de 100 Hz.

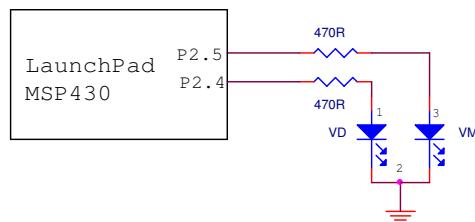


Figura 3.40. Sugestão para conexão de um *led* bicolor ao LaunchPad.

Solução: O *led* bicolor usando neste exercício é o verde e vermelho com catodo comum. Em geral, o pino do meio é o catodo comum. Basta ligá-lo à terra. O terminal 1 (o maior) é o verde e o terminal 3 é o vermelho. Se o brilho do *led* ficar muito baixo, troque os resistores de $470\ \Omega$ por resistores de $220\ \Omega$. Nos exercícios passados, escrevemos uma sub-rotina para gerar o atraso de 5 ms e as sub-rotinas para monitorar as chaves S1 e S2. Assim, nossa principal preocupação é o corpo principal do programa. A Figura 3.41 apresenta o fluxograma para a solução proposta e, logo a seguir, está sua listagem. Para que ela não ficasse muito longa, omitimos as sub-rotinas MON_S1, MON_S2, DEBOUNCE e TEMPO.

É preciso comentar alguns pontos da solução apresentada. Com a finalidade de facilitar a compreensão, criamos 4 constantes (`MODO_1`, `MODO_2`, `MODO_3` e `MODO_4`), uma para cada modo. Também foi reservada uma posição de memória denominada `MODO`, que é incrementada a cada acionamento de S1 ou S2 e que indica o modo atual. Além disso, definimos as constantes `LED_VD` (BIT4) e `LED_VM` (BIT5) para facilitar nosso acesso aos

leds. Quando é o caso de acessar os dois *leds*, essas constantes podem ser combinadas (*LED_VD* | *LED_VM*).

Para permitir que o leitor faça ensaio com diversos intervalos de tempo, foram criadas as constantes *TP5MS*, *TP10MS*, ..., *TP80MS* e para facilitar a identificação na listagem, elas estão marcadas com o comentário iniciando com “`<--`”. Veja que foram todas definidas tomando como base a constante *TP5MS*. É aborrecido localizar dentro de um longo programa uma linha para ser alterada. Para evitar isso, foi criada a constante *TP*, que sempre é igual a uma das constantes anteriores. O leitor usa *TP* para selecionar o período que deseja experimentar. Note que a sub-rotina *TEMPO*, sempre carrega essa constante *TP*. Verifique este fato na listagem da sub-rotina *TEMPO*.

As posições de memória reservadas para as variáveis *MODO*, *PS1*, *PS2*, *ES1* e *ES2* foram definidas de forma relativa. Isso facilita a alteração se, por acaso, for necessário inserir uma nova posição entre duas.

Vamos agora tratar do corpo principal da sub-rotina. Ao pensarmos na solução para piscar os *leds* nos diferentes modos, a primeira ideia que surge é a de criar um laço onde se testa o conteúdo da posição *MODO* e se executa operação correspondente. Vamos pensar um pouco mais sobre o que fazer em cada modo.

- Modo 1 → inverter o *led* verde: `XOR.B #LD_VD, &P2OUT;`
- Modo 2 → inverter o *led* vermelho: `XOR.B #LD_VM, &P2OUT;`
- Modo 3 → inverter ambos *leds*: `XOR.B #(LD_VM|LD_VM), &P2OUT` e
- Modo 4 → inverter ambos *leds*: `XOR.B #(LD_VM|LD_VM), &P2OUT;`

O leitor vai protestar, pois as operações a serem feitas nos modos 3 e 4 são idênticas. Sim! Vamos explicar como. Toda vez que o modo for alterado, é necessário apagar os dois *leds*, senão há o risco de, por exemplo, o *led* verde ficar aceso quando se passar do modo 1 para o 2. A exceção é quando se entra no modo 4, neste caso, na inicialização deve apagar um *led* e acender o outro. O leitor escolhe o *led* que fica aceso e o que fica apagado. Note que neste caso a instrução `XOR.B #(LD_VM|LD_VM), &P2OUT` vai acender o que está apagado e apagar o que está aceso, como foi pedido para o modo 4.

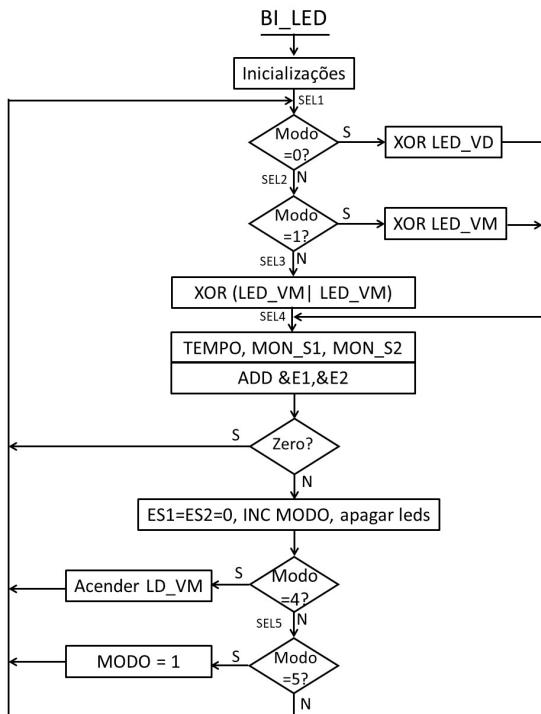


Figura 3.41. Fluxograma para a solução do exercício de piscar um led bicolor.

Outro trecho da sub-rotina que merece comentário é o que monitora as chaves. Na listagem este trecho inicia com o label SEL4. Como as chaves S1 e S2 fazem a mesma coisa, basta descobrir se uma delas foi acionada, não importa qual. Fazemos isso com a instrução ADD &ES1, &ES2. Somam-se as duas posições e se testa se o resultado é 0. Caso positivo, é porque nenhuma chave foi acionada. Caso negativo, é porque uma das foi acionada e, neste caso, precisamos incrementar o modo, mas não podemos nos esquecer de, após a consulta, zerar as posições ES1 e ES2. É claro que se a posição MODO ficar igual a 5, devemos alterá-la para 4.

Pergunta: O que pode acontecer se eliminarmos a sub-rotina que trata dos rebotes das chaves? Será que podemos diminuir o atraso que ela gera? Será que essa sub-rotina é realmente necessária? Dica: lembre-se da sub-rotina TEMPO.

Listagem da solução do ER 3.14

ER 3.14

Formatado: Inglês (Estados Unidos)

```

        CALL  #BI_LED           ;Chamar sub-rotina
        JMP   $                 ;Prender MSP
;
; Piscar led bicolor em 4 modos diferentes
; A cada acionamento de S1 ou de S2, avança para o modo seguinte
;
; Constantes:
MODO_1    .equ  1           ;1) Piscar led verde (P2.4)
MODO_2    .equ  2           ;2) Piscar led vermelho (P2.5)
MODO_3    .equ  3           ;3) Piscar leds em sincronia
MODO_4    .equ  4           ;4) Piscar leds alternadamente
LD_VD     .equ  BIT4        ;Posição do led verde na porta P2
LD_VM     .equ  BIT5        ;Posição do led vermelho na porta P2
;
TP5MS     .equ  1766         ;<-- Atraso de 5 ms
TP10MS    .equ  2*TP5MS      ;<-- Atraso de 10 ms
TP15MS    .equ  3*TP5MS      ;<-- Atraso de 15 ms
TP20MS    .equ  4*TP5MS      ;<-- Atraso de 20 ms
TP40MS    .equ  8*TP5MS      ;<-- Atraso de 40 ms
TP80MS    .equ  16*TP5MS     ;<-- Atraso de 80 ms
TP       .equ  TP15MS        ;<-- Período para Sub-rotina TEMPO
;
FECHADA   .equ  0           ;Zero representa chave fechada
ABERTA    .equ  1           ;Um representa chave aberta
DBC      .equ  10000        ;Chute para o debounce
;
; Variáveis:
MODO      .equ  0x2400       ;Modo atual
PS1       .equ  MODO+2       ;Passado da chave S1
PS2       .equ  PS1+2        ;Passado da chave S2
ES1       .equ  PS2+2        ;vai para 1 qdo S1 vai de A-->F
ES2       .equ  ES1+2        ;vai para 1 qdo S2 vai de A-->F
;
; Início da Sub-rotina
BI_LED:  CALL  #CONFIG        ;Configurar pinos
          MOV   #ABERTA,&PS1      ;Passado S1 = aberta
          MOV   #ABERTA,&PS2      ;Passado S2 = aberta
          CLR   &ES1            ;Chave S1 solta
          CLR   &ES2            ;Chave S2 solta
          MOV   #MODO_1,&MODO      ;Começar no modo 1
;
SEL1:    CMP   #MODO_1,&MODO    ;É Modo 1?
          JNZ   SEL2            ;Não é Modo 1, avançar
          XOR.B #LD_VD,&P2OUT      ;Inverter led verde
          JMP   SEL4            ;Continuar
SEL2:    CMP   #MODO_2,&MODO    ;É Modo 2?
          JNZ   SEL3            ;Não é Modo 2, avançar
          XOR.B #LD_VM,&P2OUT      ;Inverter led Vermelho
          JMP   SEL4            ;Continuar
SEL3:    XOR.B #(LD_VD | LD_VM),&P2OUT ;Inverter ambos leds

```

Formatado: Inglês (Estados Unidos)

```

;
SEL4:    CALL  #TEMPO          ;Atraso
          CALL  #MON_S1        ;Monitorar S1
          CALL  #MON_S2        ;Monitorar S2
          ADD   &ES1,&ES2       ;ES1=0 e ES2=0 ?
          JZ    SEL1           ;Sim, nenhuma chave acionada
          CLR   &ES1           ;Apagar indicar S1 acionada
          CLR   &ES2           ;Apagar indicar S2 acionada
          INC   &MODO          ;Avançar um modo
          BIC.B #(LD_VD | LD_VM),P2OUT ;Mudou Modo, apagar leds
          CMP   #MODO_4,&MODO     ;É modo 4?
          JNZ   SEL5           ;Não é modo 4, avançar
          BIS.B #LD_VM,P2OUT     ;Modo 4, acender um led
SEL5:    CMP   #5,&MODO        ;Ficou modo = 5?
          JNZ   SEL1           ;Se não, voltar ao laço
          MOV   #MODO_1,&MODO     ;Forçar para modo = 1
          JMP   SEL1           ;Voltar ao laço
;
; Colocar aqui a sub-rotina MON_S1 do ER 3.13
; Colocar aqui a sub-rotina MON_S2 do ER 3.13
; Colocar aqui a sub-rotina DEBOUNCE do ER 3.13
;
; Sub-rotina para consumir tempo
TEMPO:   MOV   #TP,R5         ;Inicializar contador 10 ms
TEMP1:   DEC   R5           ;Decrementar contador
          JNZ   TEMP1          ;Retornar quando chegar a zero
          RET
;
;
; Configurar os pinos: LED_VD(P2.4), LED_VM(P2.5), S1, S2
CONFIG:  BIS.B #(LD_VD | LD_VM),&P2DIR ;P2.4 = P2.5 = saída
          BIC.B #(LD_VD | LD_VM),&P2OUT ;VD = VM = apagado
          ;
          BIC.B #BIT1,&P2DIR      ;Configurar S1 = P2.1 = entrada
          BIS.B #BIT1,&P2REN       ;Habilitar resistor
          BIS.B #BIT1,&P2OUT       ;Selecionar pullup
          ;
          BIC.B #BIT1,&P1DIR      ;Configurar S2 = P1.1 = entrada
          BIS.B #BIT1,&P1REN       ;Habilitar resistor
          BIS.B #BIT1,&P1OUT       ;Selecionar pullup
          RET

```

ER 3.15. Este exercício é semelhante ao ER3.11, mas agora usaremos 4 *leds*. Escreva a sub-rotina CT_4SL que implementa um contador binário de 4 bits usando os 4 *leds* conectados à placa, conforme indicado na Figura 3.42. A cada acionamento de S1 (transição de aberta para fechada), o contador avança uma contagem e a cada

acionamento de S2 (transição de aberta para fechada), o contador recua uma contagem. Em outras palavras:

- S1 → incrementa o contador e
- S2 → decrementa o contador e

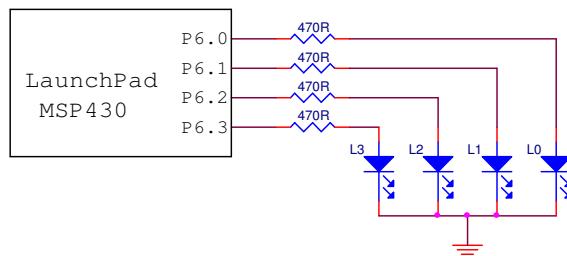


Figura 3.42. Sugestão de um contador binário com 4 leds.

Solução:

Se o brilho do led ficar muito baixo, troque os resistores de $470\ \Omega$ por resistores de $220\ \Omega$. A solução deste exercício é muito semelhante à do ER 3.11. Note que, espertamente, os leds foram conectados de acordo com a sequência dos bits da porta P6. Em outras palavras, o led L0 está ligado ao P6.0 e assim por diante até o led L3 que está conectado ao P6.3. Isto traz uma grande vantagem, pois podemos usar o registrador P6OUT como contador e incrementá-lo ou decrementá-lo.

Porém, precisamos tomar um cuidado: o que acontece com as posições que vão de P6.4 até P6.7? É claro que elas são também alteradas, seguindo a sequência do contador binário. Como essas posições não estão sendo usadas, podemos nos dar ao luxo de ignorá-las. Entretanto, isto nem sempre é verdade.

Desafio: altere o programa de forma que as posições não usadas de P6 não sejam alteradas. Tente criar a solução mais compacta possível.

Listagem da solução do ER 3.15

```
iER 3.15
    CALL #CT_4SL           ;Chamar sub-rotina
    JMP    $                 ;Prender MSP
;
; Sub-rotina : Contador de 4 bits, S1 incremente e S2 decrementa
;
; Constantes
FECHADA   .equ 0          ;Zero representa chave fechada
ABERTA    .equ 1          ;Um representa chave aberta
DBC       .equ 10000        ;Chute para o debounce
LEDS      .equ BIT3|BIT2|BIT1|BIT0 ;Leds em conjunto P6.3,2,1,0
```

Formatado: Inglês (Estados Unidos)

```

; Variáveis
VAR      .equ 0x2400          ;Área de variáveis, todas de 16 bits
PS1     .equ VAR             ;Passado da chave S1
PS2     .equ VAR+2           ;Passado da chave S2
ES1     .equ VAR+4           ;vai para 1 qdo S1 vai de A-->F
ES2     .equ VAR+6           ;vai para 1 qdo S2 vai de A-->F
;
CT_4SL:  CALL #CONFIG        ;Configurar pinos
          MOV #ABERTA,&PS1    ;Iniciar estado anterior = aberta
          MOV #ABERTA,&PS2    ;Iniciar estado anterior = aberta
          MOV #0,&CONT         ;Contador=0
LB1:    CALL #MON_S1         ;Monitorar S1
          TST &ES1            ;ES1 = 0?
          JZ  LB2              ;Salta se S1 não acionada
          INC.B &P6OUT         ;S1 acionada: Incrementar contador
          CLR  &ES1            ;ES1=0, já respondeu à chave
          ;
LB2:    CALL #MON_S2         ;Monitorar S2
          TST &ES2            ;ES2 = 0?
          JZ  LB3              ;Salta se S2 não acionada
          DEC.B &P6OUT         ;S2 acionada: Decrementar contador
          CLR  &ES2            ;ES2=0, já respondeu à chave
LB3:    JMP  LB1             ;Repetir o laço
;
; Colocar aqui a sub-rotina MON_S1 do ER 3.13
; Colocar aqui a sub-rotina MON_S2 do ER 3.13
; Colocar aqui a sub-rotina DEBOUNCE do ER 3.13
;
; Configurar os pinos: LED1, LED2, S1, S2
CONFIG:  BIS.B #LEDS,&P6DIR   ;Configurar P6.3,2,1,,0 = saída
          BIC.B #LEDS,&P6OUT   ;Apagar todos os leds
          ;
          BIC.B #BIT1,&P2DIR    ;Configurar S1 = P2.1 = entrada
          BIS.B #BIT1,&P2REN    ;Habilitar resistor
          BIS.B #BIT1,&P2OUT    ;Selecionar pullup
          ;
          BIC.B #BIT1,&P1DIR    ;Configurar S2 = P1.1 = entrada
          BIS.B #BIT1,&P1REN    ;Habilitar resistor
          BIS.B #BIT1,&P1OUT    ;Selecionar pullup
          RET

```

ER 3.16. Este exercício é semelhante ao anterior. Sua finalidade é trabalhar o conceito de tabela em *assembly*. Escreva a sub-rotina CT_7SEG que implementa um contador decimal usando um mostrador (*display*) de 7 segmentos, como ilustrado na Figura 3.43. A cada acionamento de S1 (transição de aberta para fechada), o contador avança uma contagem e a cada acionamento de S2 (transição de aberta para fechada), o contador recua uma contagem. Em outras palavras:

- S1 → incrementa o contador e
- S2 → decrementa o contador.

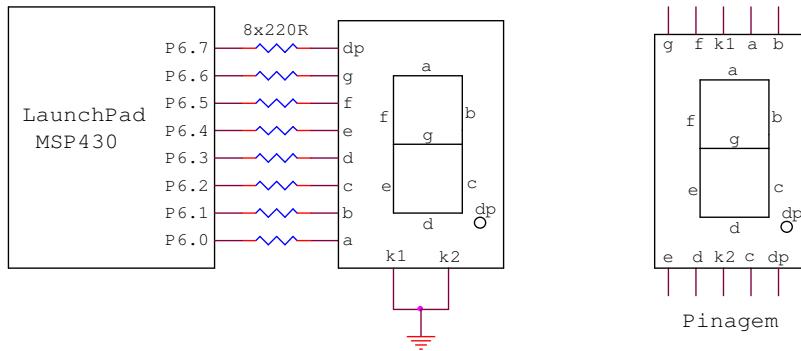


Figura 3.43. Sugestão para conexão de um mostrador de 7 segmentos Catodo Comum ao LaunchPad e sua pinagem.

Solução:

Esta solução é muito semelhante à anterior, mas agora vamos precisar de um contador em memória, que vamos chamar de `CONT`. O interesse aqui é avançar um pouco mais com a habilidade de programação em *assembly*. Na Figura 3.43 vemos que foram usados 7 bits da porta P6 para controlar os 7 segmentos do mostrador. A porta P6 foi escolhida por ela ser a única com 7 bits disponibilizados nos pinos do LaunchPad. O ponto decimal (DP) é controlado por P7.0. A listagem do programa solução está apresentada logo a seguir.

Iniciamos nossa solução construindo a Tabela 3.8 que indica, para cada número, quais segmentos que devem ser acesos. Por exemplo, para mostrar o número 0 devemos acender todos os segmentos, à exceção do segmento "g". Para mostrar o número 1, acendemos apenas os segmentos "b" e "c". Assim se constrói toda a tabela. Na coluna mais à direita está o código a ser escrito na porta P6 para gerar cada um dos números. Por exemplo, a instrução `MOV #0x66, P6OUT` faz aparecer o número 4 no mostrador.

O desafio agora é encontrar uma forma fácil de se acessar o código para cada número. A maneira mais elegante de se fazer isso é construindo uma tabela na memória de dados. Nas últimas linhas da listagem apresentada, encontra-se a seguinte linha de programa:

```
TAB_7S: .byte 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0xFF, 0x67
```

Note que essa linha armazena em sequência na memória os códigos correspondentes a cada número. O endereço de início do armazenamento está guardado na constante TAB_7S. Assim, o código do número 0 está TAB_7S+0, o código do número 1 está em TAB_7S+1, e assim por diante até o número 9 que está em TAB_7S+9. Agora ficou fácil! Para se obter o código correspondente a um número, basta somar esse número a TAB_7S e ler a posição de memória.

Tabela 3.8. Conversão dos números decimais para código 7-segmentos.

Nr	P6.6 g	P6.5 f	P6.4 e	P6.3 d	P6.2 c	P6.1 b	P6.0 a	Código
0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	1	1	0	0x06
2	1	0	1	1	0	1	1	0x5B
3	1	0	0	1	1	1	1	0x4F
4	1	1	0	0	1	1	0	0x66
5	1	1	0	1	1	0	1	0x6D
6	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	1	1	1	0x07
8	1	1	1	1	1	1	1	0xFF
9	1	1	0	0	1	1	1	0x67

A sub-rotina AT_7SEG, transcrita abaixo, tem a finalidade de atualizar o mostrador. Ela coloca em R5 o endereço da TAB_7S, adiciona a ele o valor de CONT e acessa a tabela. Note que o valor lido da tabela é escrito diretamente em P6OUT. O fato de ter todos os bits que controlam o mostrador em uma única porta, facilitou muito esta rotina. Nesta atualização, o bit 7 da porta P6OUT é sempre colocado em 0. Como ele não está sendo usado, não tem importância.

Primeira sugestão para a subrotina AT_7SEG

```
AT_7SEG:    MOV     #TAB_7S,R5 ;R5 = início da tabela
            ADD     &CONT,R5 ;Deslocar de acordo com o nr
            MOV.B @R5,&P6OUT ;Ler tabela e escrever em P6
            RET
```

Segunda sugestão para a subrotina AT_7SEG

```
AT_7SEG:    MOV     &CONT,R5 ;Nr a ser apresentado
            MOV.B TAB_7S(R5),&P6OUT ;Ler a tabela e escrever em P6
```

```
RET
```

Um outro ponto interessante neste programa é a reserva de palavras de memória para funcionar como variáveis. O programa apresentado usa as variáveis PS1, PS2, ES1, ES2 e CONT. Em programas anteriores tais espaços foram designados com a pseudo-instrução .equ que serviu para definir constantes que representavam os endereços variáveis. Isso sempre implicou em uma fase de inicialização desses espaços de memória. Na solução aqui apresentada, foi usado o recurso de carregar valores na área de dados para designar e, ao mesmo tempo, inicializar as variáveis necessárias. Vejo o trecho transcrito abaixo.

PS1	.word ABERTA	;Passado da chave S1 --> MOV #ABERTA,&PS1
PS2	.word ABERTA	;Passado da chave S2 --> MOV #ABERTA,&PS2
ES1	.word 0	;vai para 1 qdo S1 vai de A-->F
ES2	.word 0	;vai para 1 qdo S2 vai de A-->F
CONT	.word 0	;Contador binário --> MOV #0,&CONT

Listagem da solução do ER 3.16

```
;ER 3.16
        CALL  #CT_7SEG      ;Chamar sub-rotina
        JMP   $              ;Prender MSP
;
; Subrot: Contador e mostrador de 7 segmentos, S1 incr. e S2 decr.
;
; Constantes
FECHADA    .equ 0          ;Zero representa chave fechada
ABERTA     .equ 1          ;Um representa chave aberta
DBC        .equ 10000       ;Chute para o debounce
DP         .equ BIT0        ;Ponto Decimal
;
CT_7SEG:   CALL  #CONFIG      ;Configurar pinos
LB1:       CALL  #AT_7SEG;  ;Atualizar mostrador
          CALL  #MON_S1      ;Monitorar S1
          TST   &ES1          ;ES1 = 0?
          JZ    LB2          ;Salta se S1 não acionada
          CLR   &ES1          ;ES1=0, já respondeu à chave
          INC   &CONT          ;S1 acionada: Incrementar contador
          CMP   #10,&CONT       ;CONT = 10?
          JNZ   LB2          ;CONT <> 10, saltar
          CLR   &CONT          ;Chegou a 10, zerar
          ;
LB2:       CALL  #MON_S2      ;Monitorar S2
          TST   &ES2          ;ES2 = 0?
          JZ    LB1          ;Salta se S2 não acionada
          CLR   &ES2          ;ES2=0, já respondeu à chave
          DEC   &CONT          ;S2 acionada: Decrementar contador
          CMP   #-1,&CONT       ;CONT = 0xFFFF (-1)?
```

```

        JNZ    LB1      ;Se CONT <> 0xFFFF (-1), voltar laço
        MOV    #9,&CONT   ;Se CONT = 0xFFFF (-1), ir para 9
LB3:   JMP    LB1      ;Repetir o laço
;
; Sub-rotina para atualizar o mostrador de 7 segmentos
; CONT contém o número a ser apresentado, usa R5
AT_7SEG: MOV    #TAB_7S,R5 ;R5 = início da tabela
          ADD    &CONT,R5   ;Descolar de acordo com o nr
          MOV.B @R5,&P6OUT  ;Ler tabela e escrever em P6
          RET
;
; Colocar aqui a sub-rotina MON_S1 do ER 3.13
; Colocar aqui a sub-rotina MON_S2 do ER 3.13
; Colocar aqui a sub-rotina DEBOUNCE do ER 3.13
;
; Configurar os pinos: LED1, LED2, S1, S2
CONFIG:  BIS.B #0x7F,&P6DIR   ;Configurar P6.6,...,0 = saída
          BIC.B #0x7F,&P6OUT   ;Apagar todos segmentos
          ;
          BIS.B #BIT0,&P7DIR   ;Configurar DP, P7.0 = saída
          BIC.B #BIT0,&P6OUT   ;Apagar DP
          ;
          BIC.B #BIT1,&P2DIR   ;Configurar S1 = P2.1 = entrada
          BIS.B #BIT1,&P2REN   ;Habilitar resistor
          BIS.B #BIT1,&P2OUT   ;Selecionar pullup
          ;
          BIC.B #BIT1,&P1DIR   ;Configurar S2 = P1.1 = entrada
          BIS.B #BIT1,&P1REN   ;Habilitar resistor
          BIS.B #BIT1,&P1OUT   ;Selecionar pullup
          RET
;
; Segmento de dados
        .data
        ;      0   1   2   3   4   5   6   7   8   9
TAB_7S:  .byte 0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0xFF,0x67
        ;
; Reservar espaço para as demais variáveis e fazer inicialização
PS1     .word ABERTA      ;Passado da chave S1 --> MOV #ABERTA,&PS1
PS2     .word ABERTA      ;Passado da chave S2 --> MOV #ABERTA,&PS2
ES1     .word 0           ;vai para 1 qdo S1 vai de A-->F
ES2     .word 0           ;vai para 1 qdo S2 vai de A-->F
CONT    .word 0           ;Contador binário --> MOV #0,&CONT

```

Desafio: A tabela TAB_7S foi construída na memória de dados que é uma SRAM e por isso é volátil. Essa memória é carregada cada vez que o CCS entra no modo *debug*. O programa é carregado na memória *Flash*, que não é volátil. Cada vez que energizamos o LaunchPad, ele inicia rodando o programa que ficou armazenado na sua memória *Flash*. Isto significa que não precisamos do CCS para rodar um programa. Basta energizar a

Ricardo Zelenovsky, Daniel Café e Eduardo Peixoto

placa. Porém, a memória de dados não terá a `TAB_7S`, mas sim um lixo qualquer. Como garantir que a `TAB_7S` fique junto com o programa? E as variáveis que criamos e inicializamos na SRAM? Como resolver isso tudo?

ER 3.17. Este exercício é um pouco longo, mas permite ilustrar alguns recursos interessantes usados em programação *assembly*. Escreva a sub-rotina TECLADO, que armazena na memória de dados o código ASCII das teclas acionadas num teclado matricial com 16 teclas. A Figura 3.44 apresenta o teclado usado e uma sugestão para sua conexão ao LaunchPad. No conector do teclado, da forma como está apresentado na figura, as 4 primeiras conexões da esquerda para a direita são das linhas, de cima para baixo. As outras 4 conexões são para as colunas, da esquerda para a direita, como mostrado na figura abaixo.

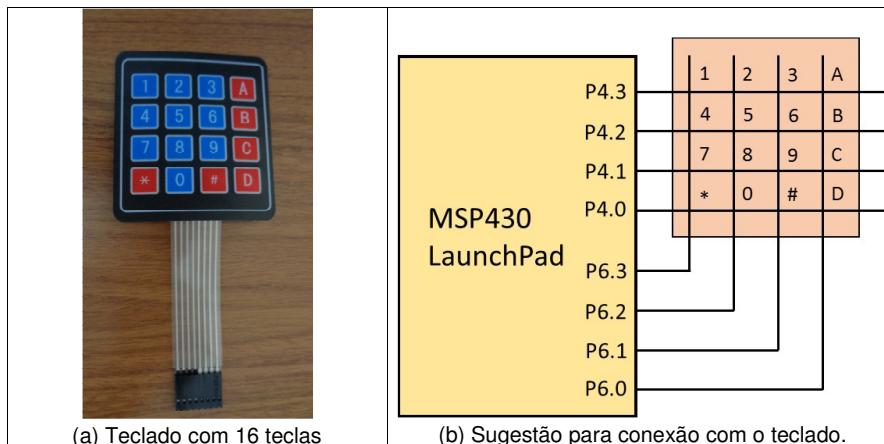


Figura 3.44. Sugestão para conexão ao LaunchPad de um teclado matricial com 16 teclas.

Solução:

Um teclado nada mais é que a reunião de uma grande quantidade de chaves. Para economizar a quantidade de portas que seria necessária, é comum organizar o teclado sob a forma matricial. A Figura 3.44 apresenta um teclado de 16 teclas conectado às portas P4 e P6. Note a existência de linhas e colunas. Em cada cruzamento de uma linha (fio) com uma coluna (outro fio) existe uma pequena chave, que quando acionada, fecha contato entre elas. Assim, a tecla “1” está no cruzamento dos fios ligados a P4.3 e P6.3, já a tecla “5” é localizada no cruzamento de P4.2 com P6.2.

O princípio para leitura do teclado é muito simples. Inicialmente, toda a P4.3,2,1,0 é colocada em nível alto. Fazendo P4.3 = 0, é possível verificar o estado das teclas 1, 2, 3 e A. Por exemplo, se a tecla 3 estiver pressionada, então apenas o pino P6.1 estará em nível baixo. Em seguida é feito P4.3 = 1 e P4.2 = 0, agora verifica-se o estado das teclas 4, 5, 6 e B. Assim por diante. Pelos pinos P4.3, P4.2, P4.1 e P4.0 é feita a varredura (*scan*) e o resultado é lido por P6.3, P6.2, P6.1 e P6.0. A Tabela 3.9 apresenta um resumo com os códigos de varredura e as teclas varridas. Deve-se construir uma rotina que, de tempos em tempos, faça uma varredura do teclado para verificar se alguma tecla foi acionada.

Tabela 3.9: Códigos de varredura para o teclado proposto

P4.3	P4.2	P4.1	P4.0	Teclas varridas
0	1	1	1	1, 2, 3, A
1	0	1	1	4, 5, 6, B
1	1	0	1	7, 8, 9, C
1	1	1	0	*, 0, #, D

A figura a seguir apresenta exemplos de varredura do teclado, usando a sequência apresentada na Tabela 3.9. Para saber qual tecla foi acionada, basta verificar onde surgiu o zero. Se não for lido um valor zero, então é porque nenhuma tecla foi acionada. Por outro lado, se forem lidos dois ou mais zeros, é porque duas ou mais teclas estão acionadas. Nesse caso é costume rejeitar a leitura. É preciso tomar cuidado para não se detectar múltiplas vezes o acionamento de uma mesma tecla. Por isso, para que se aceite uma nova tecla, o teclado precisa obrigatoriamente passar pelo estado de nenhuma tecla acionada.

P4.3:0 P6.3:0		P4.3:0 P6.3:0	
3	2	1	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	1
1	1	1	1
Tecla 6		Tecla 8	

P4.3:0	P6.3:0	P4.3:0	P6.3:0
3 2 1 0	3 2 1 0	3 2 1 0	3 2 1 0
0 1 1 1	1 1 1 1	0 1 1 1	1 0 1 1
1 0 1 1	1 1 1 1	1 0 1 1	1 1 0 1
1 1 0 1	1 1 1 1	1 1 0 1	1 1 1 1
1 1 1 0	1 1 1 1	1 1 1 0	1 1 1 1
Nenhuma tecla acionada		Teclas 2 e 6	

Figura 3.45. Quatro exemplos de códigos de varredura do teclado.

A listagem solução está apresentada mais adiante. Ela é relativamente complexa e vamos analisar cada sub-rotina que a compõe em separado. São 3 sub-rotinas, que enumeramos a seguir.

- **TECLADO** → faz inicializações e fica num laço, escrevendo a partir da posição BUFFER o código ASCII das teclas acionadas
- **TEC_ZERO** → analisa a varredura recebida e armazena a tecla acionada na posição de memória denominada TECLA e, por fim,
- **TEC_SCAN** → faz a varredura do teclado e retorna o resultado em R5.

Vamos começar com a sub-rotina TEC_SCAN, responsável por colocar os códigos de varredura na porta P4 e ler os resultados na porta P6. Usamos apenas 4 bits desta porta P4, os demais bits serão ignorados, pois eles não estão sendo usados. Como são 4 varreduras e cada varredura retorna 4 bits, vamos usar os 16 bits do registrador R5. A seguir está o resumo da sub-rotina TEC_SCAN. O fluxograma e a explicação estão apresentados na Figura 3.46. A relação entre cada bit de R5, que recebe o resultado das 4 varreduras, e as teclas está na Figura 3.47.

Sub-rotina TEC_SCAN

Recebe: nada

Retorna: R5 = resultado da varredura

Usa: R6 = varreduras

R7 = auxiliar na montagem do código resultante

R8 = contador

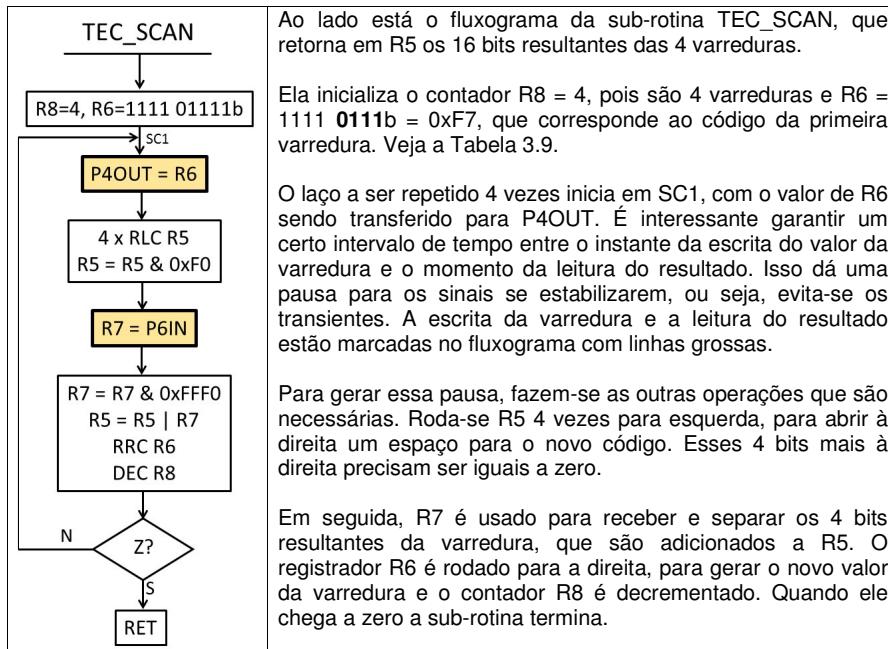


Figura 3.46. Fluxograma da sub-rotina TEC_SCAN e sua explicação.

Varredura	0111				1011				1101				1110			
R5 (bits)	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Teclas	1	2	3	A	4	5	6	B	7	8	9	C	*	0	#	D

Figura 3.47. Relação entre o valor de cada varredura e a posição de acionamento das teclas de acordo com os bits de R5.

Passamos agora à análise da sub-rotina TEC_ZERO, que recebe em R5 o resultado da varredura e atualiza a posição de memória TECLA com o código ASCII da tecla que foi acionada. O fluxograma correspondente está apresentado na Figura 3.48. Essa sub-rotina precisa ainda indicar a situação de nenhuma tecla acionada e a situação de tecla inválida (duas ou mais teclas acionadas). Para facilitar essas indicações são criados dois códigos extras: T_NADA e T_INV. Assim, os possíveis valores da posição de memória TECLA são:

- TECLA = 0 (T_NADA), nenhuma tecla acionada;
- TECLA = 1 (T_INV), duas ou mais teclas acionadas e
- TECLA = 1(0x31), 2 (0x32), ..., 9(0x39), A(0x41), ..., D(0x44), *(0x2A) e #(0x23).

Usando R8 como contador, a sub-rotina percorre R5, da esquerda para a direita, contando quantos bits estão em zero. Toda vez que encontra um *bit* igual a zero, incrementa R7 e o valor do contador R8 é copiado para R6. Ao finalizar temos três possíveis resultados para R7:

- R7 = 0 → nenhuma tecla açãoada;
- R7 = 1 → foi açãoada apenas uma tecla e sua posição está marcada em R6 e
- R7 > 1 → tecla inválida pois foram açãoadas duas ou mais teclas.

Para facilitar a tradução da posição do *bit* igual a 0 (valor de R6) para o código ASCII da tecla açãoada, foi usada uma tabela de 17 posições, mostrada a seguir. Da forma como foi construída a sub-rotina TEC_ZERO, R6 marca a posição do *bit* igual zero com valores de 1 até 16. Por isso, R6 = 0 indica que nenhuma tecla foi açãoada. Assim, basta a instrução `MOV.B TAB_TEC(R6), &TECLA` para escrever na posição de memória TECLA o código ASCII da tecla açãoada.

```
TAB_TEC: .byte T_NADA, "D#0*C987B654A321"
```

Especificação da sub-rotina em questão:

Sub-rotina TEC_ZERO

Recebe: R5 = código resultante da varredura

Retorna: TECLA = código ASCII da tecla açãoada

Usa: R6 = posição do bit = 0

R7 = quantidade de bits iguais a zero

R8 = contar os 16 bits de R5

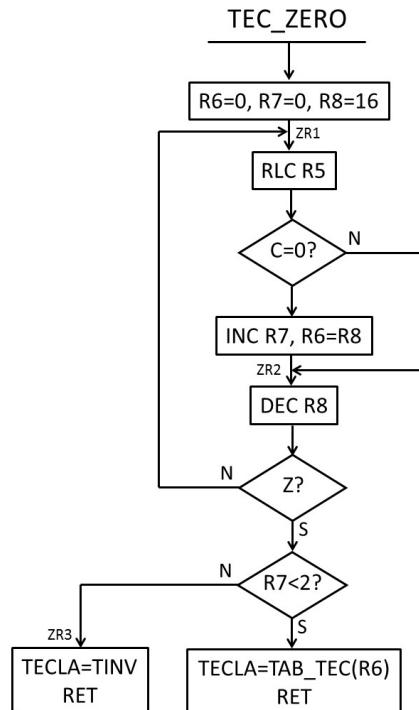


Figura 3.48. Fluxograma da sub-rotina TEC_ZERO.

Passemos agora para a análise da sub-rotina TECLADO. Ela recebe na posição TECLA o código da tecla acionada e, se for válida, a escreve em sequência a partir da posição BUFFER. Existem 3 classes de teclas: T_NADA, T_INV ou o ASCII (da tecla acionada). Essa sub-rotina precisa ser inteligente porque o ato de apertar e liberar uma tecla é lento e há o risco de se aceitar várias vezes o acionamento da mesma tecla. Por isso, uma tecla só é aceita se o teclado antes passou pelo estado de nenhuma tecla acionada. Isto significa que a rotina precisa se lembrar do último estado do teclado. Além disso, a condição de teclado inválido também precisa ser tratada de forma coerente.

Para manejá-las com todas essas situações, o mais simples é empregarmos uma Máquina de Estados. São 3 os estados possíveis:

- E0: teclado solto, nenhuma tecla acionada;
- E1: teclado inválido, duas ou mais teclas acionadas e
- E2: tecla válida, foi armazenada no BUFFER.

As transições dessa máquina estão apresentadas na Tabela 3.10. Enquanto o teclado permanecer solto, a máquina fica no estado E0. Note que uma tecla é colocada no BUFFER (tecla aceita) somente na transição de E0 para E2. Se chegar uma tecla inválida, a máquina vai para o estado E1 e só sai dele na condição de nenhuma tecla acionada. De E2 a máquina volta para E0 (teclado solto) ou vai para E1(teclado inválido). A última coluna da Tabela de Transição de Estados não é usada pois a condição de nenhuma tecla e teclado inválido não pode existir. A seguir está a Figura 3.49 que traz o Diagrama de Estados correspondente à Tabela de Estados. A Tabela de Estados e o Diagrama de Estados representam a mesma máquina. O leitor usa a que melhor se adaptar.

Tabela 3.10: Tabela de Transição de Estados para a lógica de tratamento das teclas.

Descrição	Estado	Entradas (N=Nada e I=Inválido)			
		$\bar{N} \cdot \bar{I}$	$N \cdot \bar{I}$	$\bar{N} \cdot I$	$N \cdot I$
Teclado solto	E0	E2/tecla OK	E0	E1	-
Tecla inválida	E1	E1	E0	E1	-
Tecla válida	E2	E2	E0	E1	-

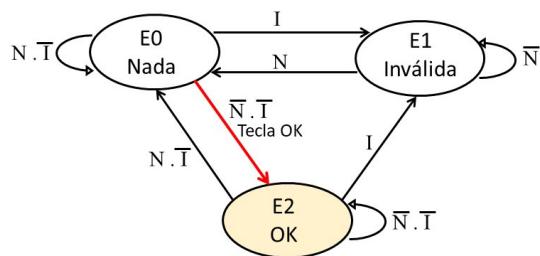


Figura 3.49. Diagrama de estados para a lógica de tratamento das teclas.

A construção de uma Máquina de Estados em assembly precisa seguir a Tabela de Estados ou o Diagrama de Estados projetado. A forma genérica desta máquina está apresentada no fluxograma da Figura 3.50. Começa-se por identificar o estado presente, o que é feito pela coluna principal com o teste para ver o valor da variável Estado. Para cada valor desta variável, desvia-se para um trecho que faz o tratamento referente ao estado presente. Basicamente, este tratamento consiste em verificar as entradas e, de acordo com os valores dessas entradas, geram-se as saídas, atualiza-se a variável Estado para seu novo valor (próximo estado) e fecha-se o laço.

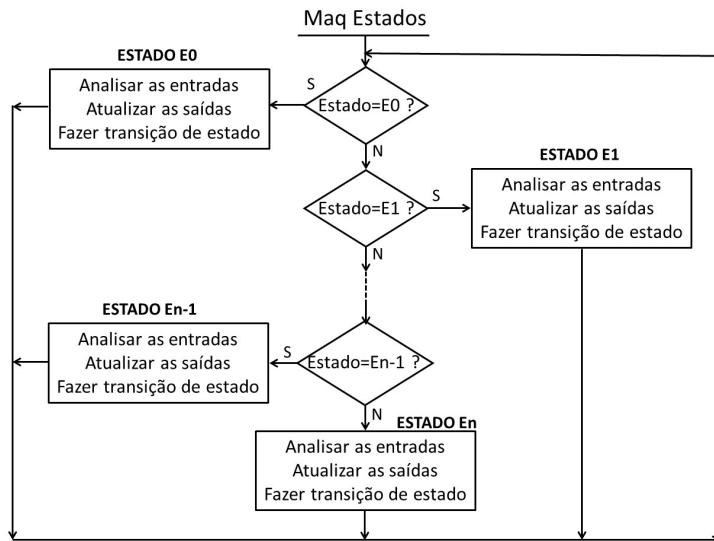


Figura 3.50. Fluxograma para uma Máquina de Estados.

Existem dois modelos de Máquinas de estado. O primeiro, denominado de Modelo Moore, gera as saídas específicas enquanto a máquina permanecer num determinado estado. Neste tipo de máquina, as saídas dependem apenas dos estados. O outro modelo, denominado de Modelo Mealy, gera as saídas por ocasião das transições de estado. Neste tipo de máquina, as saídas dependem dos estados e das entradas. O termo saída é interpretado de forma bem amplo. Saída pode ser a alteração de um *bit* de GPIO, pode ser a escrita em memória, como foi nosso caso, ou ainda a chamada de uma determinada sub-rotina, ou ainda, uma outra ação de *software* que se faça necessária.

No caso da sub-rotina TECLADO, foi usado o modelo Mealy e a saída foi caracterizada pela ação de escrever a variável `TECLA` no `BUFFER`. Note que a tecla só foi aceita quando a máquina transitou do estado `E0` (teclado solto) para o estado `E2` (tecla válida). Enquanto permaneceu no estado `E2`, nenhuma tecla mais foi escrita no `BUFFER`. Recomenda-se uma análise cuidadosa do fluxograma da Figura 3.50.

Mais adiante está a listagem do programa TECLADO. Para facilitar seu entendimento, foram criadas as constantes para os estados: `EST_0`, `EST_1` e `EST_2`. Pode parecer irrelevante, mas isso facilita a leitura do programa. Toda vez que essas constantes forem envolvidas é porque se está trabalhando com estado da máquina. As constantes `T_NADA` e `T_INV` são usadas para facilitar a indicação de que o teclado está solto ou que está na condição de inválido.

Veja que no final da listagem estão reservadas posições para o `ESTADO`, `TECLA`, `TAB_TEC`, `CONT` e `BUFFER`. Note que essas posições já são inicializadas. Isto dispensaria as três linhas listadas abaixo. Entretanto, essas posições só são atualizadas quando se carrega o programa na memória. Assim, para permitir que o usuário use a opção de “Soft Reset” do CCS para rodar o programa várias vezes, essas três linhas foram adicionadas.

```
MOV.B #T_NADA,&TECLA      ;TECLA = NADA
MOV.B #EST_0,&ESTADO       ;ESTADO = EST_0
CLR.B &CONT                 ;Contador de teclas = 0
```

As teclas acionadas são escritas a partir do endereço `BUFFER`, usando como incremento o valor da posição `CONT`. Como `CONT` tem apenas 8 bits, a quantidade máxima de teclas que se pode receber sem sobrescrever as anteriores é 256. É importante que `BUFFER` ocupe a última posição.

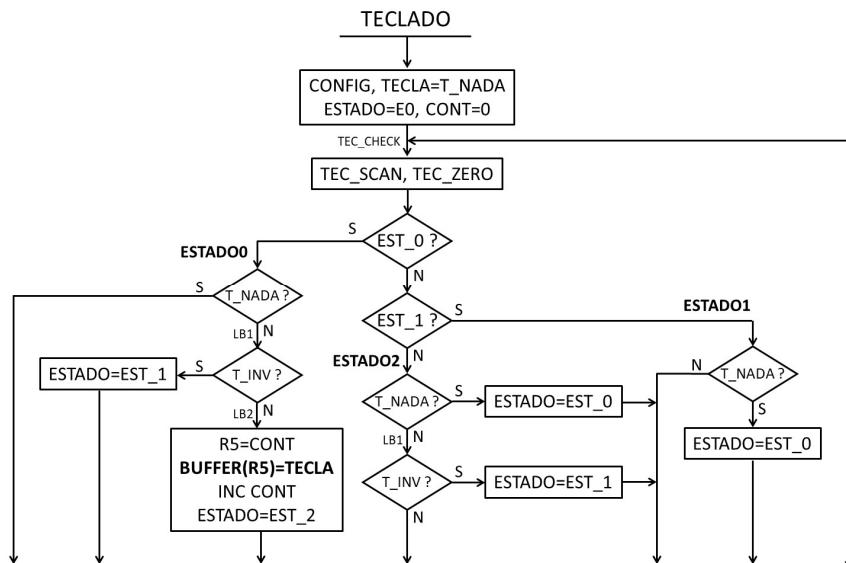


Figura 3.50. Fluxograma da sub-rotina `TECLADO`.

Listagem da solução do ER 3.17

```
;ER 3.17
CALL  #TECLADO      ;Chamar sub-rotina
JMP   $              ;Prender MSP (nunca chega aqui)
```

```

;
; TECLADO: Ler teclado matricial
; Variáveis: ESTADO, TECLA, CONT
; Retorna: BUFFER = código ASCII das teclas acionadas;
; Estados para máquina do teclado
EST_0 .equ 0      ;NADA TECLADO
EST_1 .equ 1      ;TECLA INVÁLIDA
EST_2 .equ 2      ;TECLA OK
;
T_NADA .equ 0      ;Teclado solto
T_INV   .equ 1      ;Teclado inválido
;
TECLADO:
    CALL  #CONFIG          ;Configurar pinos
    MOV.B #T_NADA,&TECLA    ;TECLA = NADA
    MOV.B #EST_0,&ESTADO     ;ESTADO = EST_0
    CLR.B &CONT            ;Contador de teclas = 0
;
TEC_CHECK:
    CALL  #TEC_SCAN        ;Fazer scan do teclado
    CALL  #TEC_ZERO         ;Localizar tecla acionada
    CMP.B #EST_0,&ESTADO     ;ESTADO = EST_0
    JZ    ESTADO_0           ;Sim, vai para ESTADO_0
    CMP.B #EST_1,&ESTADO     ;ESTADO = EST_1
    JZ    ESTADO_1           ;Sim, vai para ESTADO_1
ESTADO_2:
    CMP.B #T_NADA,&TECLA    ;TECLA = NADA?
    JNZ   LB1                ;Não, vai para LB1
    MOV.B #EST_0,&ESTADO     ;Sim, ESTADO = EST_0
    JMP    TEC_CHECK          ;Volta ao laço
LB1:
    CMP.B #T_INV,&TECLA     ;TECLA = Inválida?
    JNZ   TEC_CHECK          ;Não, volta ao laço
    MOV.B #EST_1,&ESTADO     ;Sim, ESTADO = EST_1
    JMP    TEC_CHECK          ;Vai para o laço
;
ESTADO_1:
    CMP.B #T_NADA,&TECLA    ;TECLA = NADA?
    JNZ   TEC_CHECK          ;Não, volta ao laço
    MOV.B #EST_0,&ESTADO     ;Sim, ESTADO = EST_0
    JMP    TEC_CHECK          ;Volta ao laço
;
ESTADO_0:
    CMP.B #T_NADA,&TECLA    ;TECLA = NADA?
    JZ    TEC_CHECK          ;Sim, volta ao laço
    CMP.B #T_INV,&TECLA     ;TECLA = Inválida?
    JNZ   LB2                ;Não, vai para frente
    MOV.B #EST_1,&ESTADO     ;Sim, ESTADO = EST_1
    JMP    TEC_CHECK          ;Volta ao laço
LB2:
    MOV.B &CONT,R5           ;Chegou tecla válida
    MOV.B &TECLA,BUFFER (R5)  ;Armazenar no buffer

```

```

INC.B &CONT           ;Incrementar contador
MOV.B #EST_2,&ESTADO   ;ESTADO = EST_2
JMP    TEC_CHECK       ;Volta ao laço
;
; Verifica se só tem um zero
; R5 --> código da varredura
; R6 --> posição do primeiro bit = 0
; R7 --> Contar quantidade de bits = 0
; R8 --> Contador = 16
TEC_ZERO:
    CLR   R7           ;Zerar contador de bits = 0
    CLR   R6           ;Zerar posição primeiro bit = 0
    MOV   #16,R8        ;Serão 16 repetições
ZR1:  RLC   R5           ;Rodar R5 com Carry
    JC    ZR2          ;Saltar se Carry = 1?
    INC   R7           ;Cy = 0 --> incrementar contador de bit = 0
    MOV   R8,R6        ;R6 = posição do bit = 0
ZR2:  DEC   R8           ;Decrementar contador
    JNZ   ZR1          ;Contador <> 0, repetir
    CMP   #2,R7        ;R7 - 2 (buscar por R7 = 0 ou 1)
    JHS   ZR3          ;Se R7 >= 2, saltar
    MOV.B TAB_TEC(R6),&TECLA ;Ler código da tecla acionada
    RET
ZR3:  MOV.B #T_INV,&TECLA   ;Mais de um bit = 0, inválido
    RET
;
; TEC_SCAN: Retornar o código de varredura em R5
; R6 = Varreduras
; R7 = Recebe código parcial
; R8 = Contador
TEC_SCAN:
    MOV   #4,R8         ;4 scans
    MOV   #0xF7,R6        ;Primeira varredura = 1111 0111
SC1:  MOV.B R6,&P4OUT    ;P4 = varredura
    RLC   R5           ;;;1) Deslocar R5 para esquerda
    RLC   R5           ;;;2) 4 vezes
    RLC   R5           ;;;3) para abrir espaço
    RLC   R5           ;;;4) para novo código
    BIC   #0xF,R5        ;Apagar 4 bits da direita
    MOV.B &P6IN,R7      ;Ler código de varredura parcial
    BIC   #0xFFFF0,R7     ;Separar 4 bits da direita = código parcial
    BIS   R7,R5          ;Transferir código parcial para R5
    RRC   R6           ;Gerar nova varredura rodando R6
    DEC   R8           ;Decrementar contador
    JNZ   SC1          ;Se contador <> 0, repetir
    RET
;
; Configurar os pinos: LED1, LED2, S1, S2
CONFIG: ;Linhas = saída, códigos de scan
        BIS.B #0xF,&P4DIR ;Configurar P6.3:0 = saída

```

```
BIS.B #0xF,&P4OUT ;P6.3:0 = 1111
;
; Colunas = entrada, scan code, resultado
BIC.B #0xF,&P6DIR ;Configurar P4.3:0 = entrada
BIS.B #0xF,&P6REN ;Habilitar resistor de P4.3:0
BIS.B #0xF,&P6OUT ;Habilitar pullup de P4.3:0
RET
;
; Segmento de dados
.data
ESTADO: .byte EST_0
TECLA: .byte T_NADA
TAB_TEC: .byte T_NADA, "D#0*C987B654A321"
CONT: .byte 0 ;Contador de teclas (limite = 256)
BUFFER : .byte 0 ;Buffer para o que for teclado
```

3.9. Exercícios Propostos

A seguir são propostos diversos exercícios. Alguns deles são meras ampliações de algum dos exercícios resolvidos, enquanto outros são completamente novos. É fortemente recomendado que o leitor, após escrever seu programa, o simule usando o Code Composer Studio (CCS) e a placa MSP430 F5529 LaunchPad. Sempre separe sua solução (em geral, uma sub-rotina) do ambiente usado para a verificação. Em muitos casos, será necessário corrigir erros descobertos durante a fase de verificação.

EP 3.1. Escreva a sub-rotina PS_VM_VD que fica presa em um laço, piscando o LED1 (P1.0) que é o vermelho e o LED2 (P4.7) que é o verde. Os *leds* acendem e apagam ao mesmo tempo, ou seja, sincronizados. Crie uma sub-rotina TEMPO, para garantir que o piscar fique numa frequência passível de ser observada.

EP 3.2. Escreva a sub-rotina PA_VM_VD que fica presa em um laço, piscando o LED1 (P1.0) que é o vermelho e o LED2 (P4.7) que é o verde. Os *leds* acendem e apagam de forma alternada. Crie uma sub-rotina TEMPO, para garantir que o piscar fique numa frequência passível de ser observada.

EP 3.3. Altere a sub-rotina PA_VM_VD do exercício anterior, de forma a que o piscar dos *leds* ocorra na frequência de 5 Hz (100 ms aceso e 100 ms apagado) para cada um, mantendo o piscar alternado.

EP 3.4. Escreva a sub-rotina P1_VM que acenda e apague o LED1 de acordo com o padrão repetitivo mostrado na Figura 3.51.



Figura 3.51. Padrão repetitivo a ser usado para comandar o LED1.

EP 3.5. Escreva a sub-rotina P1_VM_VD que acenda e apaga os *leds* LED1 e LED2 de acordo com os padrões repetitivos mostrados na Figura 3.52.

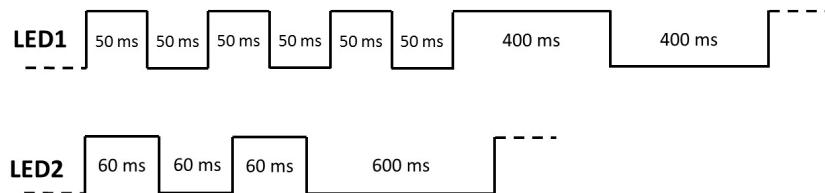


Figura 3.52. Padrão repetitivo a ser usado para comandar os leds LED1 e LED2.

EP 3.6. Este exercício é semelhante ao ER 3.6. Escreva a sub-rotina P1_SL que mantém aceso o LED1 (P1.0), que é o vermelho, enquanto a chave S1 (P2.1) permanecer açãoada e o LED2 (P4.7), que é o verde, enquanto a chave S2 (P1.1) permanecer açãoada. Se as duas chaves forem açãoadas ao mesmo tempo, os *leds* devem ser apagados.

EP 3.7. Este exercício é semelhante ao EP 3.6. Escreva a sub-rotina P2_SL que mantém aceso o LED1 (P1.0), que é o vermelho, enquanto a chave S1 (P2.1) permanecer açãoada e o LED2 (P4.7), que é o verde, enquanto a chave S2 (P1.1) permanecer açãoada. Se as duas chaves forem açãoadas ao mesmo tempo, os *leds* devem piscar de forma alternada na frequência de 10 Hz (50 ms aceso e 50 ms apagado).

EP 3.8. No ER 3.9, os rebotes são eliminados graças a uma espera determinada empiricamente. Vamos pedir para resolver esse exercício, mas com uma abordagem diferente. A ideia é propor uma rotina que verifica a presença de rebotes e retorna quando eles deixam de estar presentes. Os dois fluxogramas abaixo apresentam a ideia. Note que são necessárias duas sub-rotinas, uma para cada flanco, e que elas só têm utilidade quando se deseja prender a execução aguardando o açãoamento de uma chave. A constante RBT deve ser determinado empiricamente. Então, repita o ER 3.9 usando esta nova proposta para eliminar os rebotes.

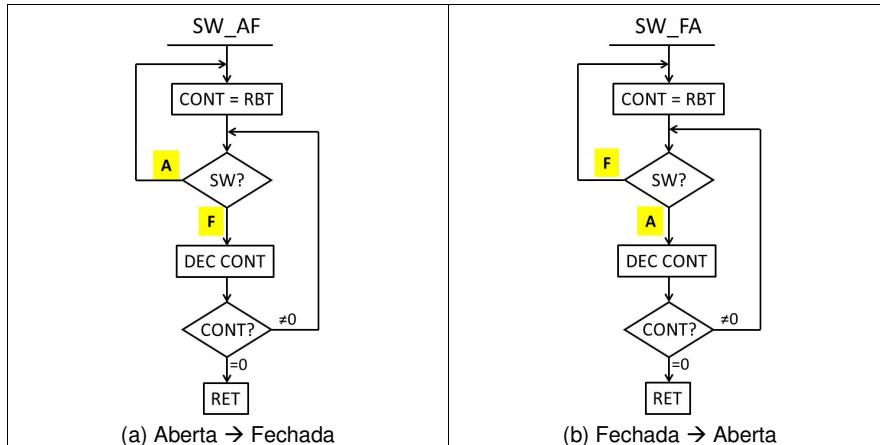


Figura 3.53. Sugestão de duas rotinas para eliminar os rebotes: (a) ao acionar a chave e (b) ao liberar a chave.

EP 3.9. Este exercício é semelhante aos ER 3.8 e ER 3.11. Escreva a sub-rotina CT1_SL que implementa um contador binário com os dois *leds* da placa. Cada acionamento de S1 (transição de aberta para fechada) pode incrementar (+1) ou decrementar (-1) o valor do contador, de acordo com o estado da chave S2, como mostrado na tabela abaixo. Faça o tratamento dos rebotes.

Tabela 3.11. Comando para o Contador de 2 bits construído com LED1 e LED2.

Contagem	S1	S2
+1	A → F	A
-1	A → F	F

A = Aberta e F = Fechada

EP 3.10. Este exercício é semelhante ao EP 3.9. Escreva a sub-rotina CT2_SL que implementa um contador binário com os dois *leds* da placa. Se a chave S2 está aberta, o contador é incrementado a cada acionamento de S1 (transição de aberta para fechada), porém, se a chave S2 está fechada, o contador é incrementado a cada liberação de S1 (transição de fechada para aberta), como mostrado na Tabela abaixo. Faça o tratamento dos rebotes.

Tabela 3.12. Comando para o Contador de 2 bits construído com LED1 e LED2.

Contagem	S1	S2
+1	A → F	A
+1	F → A	F

A = Aberta e F = Fechada

EP 3.11. Este exercício é semelhante ao EP 3.9. Escreva a sub-rotina CT3_SL que implementa um contador binário com os dois *leds* da placa. O contador só é incrementado ou decrementado quando as duas chaves são acionadas. Se S1 foi acionada antes de S2, o contador é incrementado e se S2 foi acionada antes de S1, o contador é decrementado.

EP 3.12. Este exercício é semelhante ao EP 3.9. Escreva a sub-rotina CT4_SL que implementa um contador binário com os dois *leds* da placa. Vamos introduzir o seguinte controle: enquanto a chave que foi acionada primeiro for mantida pressionada, os acionamentos da segunda chave alterarão o contador (incrementando ou decrementando).

- S2 é acionada primeiro e depois cada acionamento de S1 incrementa o contador e
- S1 é acionada primeiro e depois cada acionamento de S2 decrementa o contador.

EP 3.13. Este exercício é semelhante ao EP 3.9. Escreva a sub-rotina CT5_SL que implementa um contador binário com os dois *leds* da placa. Vamos introduzir o seguinte controle: as chaves precisam passar obrigatoriamente pelo estado de ambas abertas para que se aceite uma nova alteração do contador (incremento ou decremento).

EP 3.14. Este exercício é semelhante ao ER 3.12 e pede para escrever a sub-rotina JOGO1, que lembra uma disputa entre dois jogadores onde vence quem primeiro acionar sua chave. O início da disputa é caracterizado pelo padrão indicado na Figura 3.54. Ganha aquele que acionar primeiro sua chave. Se for a chave S1, o LED1 acende. Se for a chave S2, o LED2 acende. O jogo só recomeça quando as duas chaves estiverem soltas e os *leds* apagados.

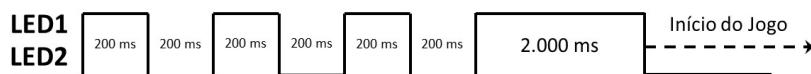


Figura 3.54. Padrão de acendimento dos leds para caracterizar o início do jogo.

EP 3.15. Vamos aperfeiçoar o exercício anterior. Note que se um dos jogadores acionar sua chave um pouco antes da largada (“queimar a largada”), ele acaba ganhando. Não há qualquer tipo de proteção contra isso. Então, escreva a sub-rotina JOGO2, semelhante à JOGO1, mas com um recurso extra para detectar o acionamento indevido de uma das chaves. O acionamento indevido deve ser indicado com o pisca-pisca de um dos *leds* na frequência de 2,5 Hz (200 ms aceso e 200 ms apagado). O LED1 piscando indica que a chave S1 “queimou a largada”. Isso vale para o LED2 e a chave S2. Esse pisca-pisca continua enquanto uma das chaves estiver acionada. Somente quando as duas chaves estiverem soltas é que os *leds* são apagados e o jogo inicia com o padrão da Figura 3.54.

EP 3.16. O ER 3.13 apresentou um PWM com período 20 ms para controlar o brilho do LED1 (P1.0). Ele possuía apenas valores: 0%, 25%, 50%, 75% e 100%. Na solução apresentada foi notado que é o LED1 ficava levemente aceso mesmo com PWM = 0%. Escreve a sub-rotina PWM2, que garante que o *led* fique realmente apagado quando o fator de carga é 0% e que faça uso da dupla *bufferização*.

EP 3.17. O ER 3.13 apresentou um PWM para o controle do brilho do LED1. Escreva a sub-rotina PWM3 que controla o brilho dos dois *leds* da placa, mas de forma complementar. Por exemplo, quando o LED1 estiver com brilho de 25%, o LED2 estará com brilho de 75%. Garanta que o *led* esteja apagado com PWM = 0% e use a dupla bufferização.

EP 3.18. Este exercício é um aperfeiçoamento do EP 3.16. Escreva a sub-rotina PWM4, que constrói um PWM de período 20 ms para controlar o brilho do LED1 (P1.0) que é o vermelho, em 10 passos, indo de 0% até 100%.

EP 3.19. Este exercício é semelhante ao ER 3.15 que usava um contador de 4 bits construído com 4 *leds*. No presente exercício, não temos um contador, mas uma situação em que sempre há um *led* aceso. Escreva a sub-rotina RD_4SL que, de forma circular, move o *led* aceso para a esquerda (SL) ou para a direita (SR). Use a referência da Figura 3.55. Um acionamento de S1 (transição de aberta para fechada) corresponde ao giro para a direita e um acionamento de S2 (transição de aberta para fechada), corresponde a um giro para a esquerda. O programa parte com o *led* L0 aceso.

- S1 → (SR) girar para a direita
- S2 → (SL) girar para a esquerda

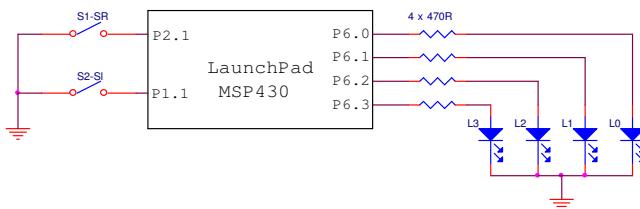


Figura 3.55. Sugestão para conexão de 4 leds ao LaunchPad.

EP 3.20. Este exercício é semelhante ao ER 3.16. Vamos usar o ponto decimal (dp) para indicar que o contador ultrapassou o 9 e voltou ao 0. Temos então 10 contagens com o ponto decimal apagado e depois 10 contagens com o ponto decimal aceso. No fim das contas, construímos um contador módulo 20.

EP 3.21. Este exercício é semelhante ao ER 3.16. A única diferença é que o contador agora é hexadecimal. O leitor deverá sugerir um padrão para indicar os valores hexadecimais (0 até 15) num *display* de 7 segmentos. Busque por sugestões na Internet.

EP 3.22. Este exercício é semelhante ao ER 3.16, mas agora temos apenas um segmento aceso e vamos girá-lo no sentido horário ou no sentido anti-horário, como mostrado na Figura 3.56. Note que o segmento “g” e o ponto decimal “dp” sempre estão apagados.

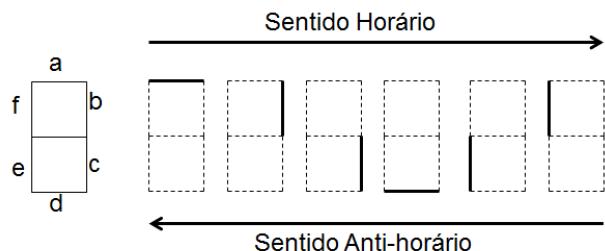


Figura 3.56. Referência para o giro do led aceso.

A Figura 3.57 apresenta o circuito e indica a ação que cada chave deve realizar. É claro que as chaves só devem atuar ao passar do estado aberta para fechada e os rebotes devem ser eliminados.

Dica: escreva duas sub-rotinas abaixo.

GHO → apaga o segmento que está aceso e acende o próximo no sentido horário e
GAH → apaga o segmento que está aceso e acende o próximo no sentido anti-horário.

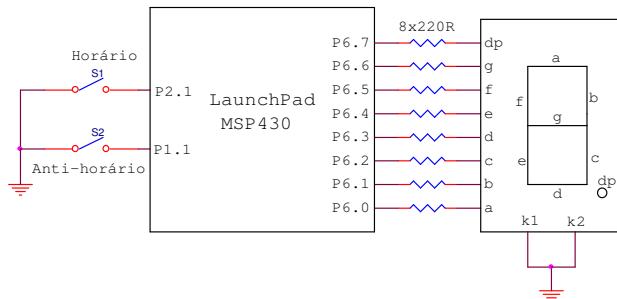


Figura 3.57. Sugestão para conexão de um display de 7 segmentos ao LaunchPad.

EP 3.23. Este exercício é semelhante ao ER 3.16, que usa um mostrador (*display*) de 7 segmentos. O consumo desse tipo de mostrador é elevado. Cada segmento gasta perto de 15 mA. Se os 7 segmentos estão acesos ao mesmo tempo, o consumo total é de 105 mA. Um truque para reduzir esse consumo e piscar o mostrador numa frequência acima da percepção do olho humano. Note na Figura 3.58 que o transistor Q, quando cortado apaga o mostrador. Pino P7.4 = 1 liga o transistor e P7.4 = 0 desliga o transistor.

Escreva a sub-rotina CT_7SEG1 que implementa um contador decimal usando um mostrador (*display*) de 7 segmentos, como ilustrado na Figura 3.58, que deve piscar na frequência de 100 Hz. A cada acionamento de S1 (transição de aberta para fechada), o contador avança uma contagem e a cada acionamento de S2 (transição de aberta para fechada), o contador recua uma contagem. Em outras palavras:

- S1 → incrementa o contador e
- S2 → decrementa o contador.

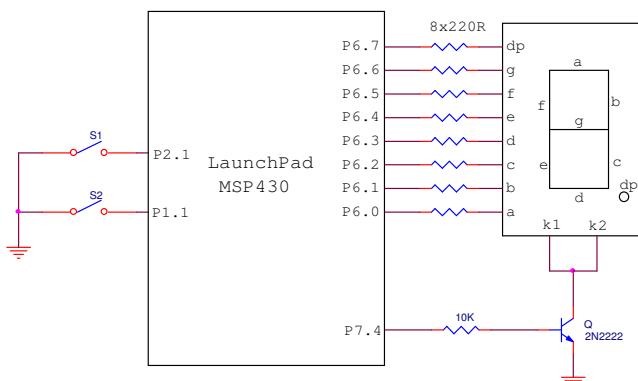


Figura 3.58. Uso do transistor Q para permitir apagar o mostrador.

EP 3.24. O EP 3.23 fez uso do transistor Q para acender ou apagar o mostrador. Existe uma forma mais simples de se apagar esse mostrador: basta escrever 0 em todos os bits da porta P6. Escreva a sub-rotina CT_7SEG2 que implementa um contador decimal usando um mostrador (*display*) de 7 segmentos, como ilustrado na Figura 3.57 (EP 3.22), que deve piscar na frequência de 100 Hz. A cada acionamento de S1 (transição de aberta para fechada), o contador avança uma contagem e a cada acionamento de S2 (transição de aberta para fechada), o contador recua uma contagem. Em outras palavras:

- S1 → incrementa o contador
- S2 → decrementa o contador.

EP 3.25. O ER 3.17 mostrou como se deve usar um teclado matricial. O tratamento de rebotes foi, de certa forma, ignorado. Como as sub-rotinas para tratamento deste teclado matricial introduzem atrasos apreciáveis, talvez, na grande maioria dos casos os rebotes não prejudiquem. Entretanto, seria bom colocar um pequeno cuidado neste sentido. A sugestão é rejeitar os rebotes com a exigência de que duas varreduras seguidas feitas pela sub-rotina TEC_SCAN retornem o mesmo resultado. Caso não sejam idênticas, as varreduras são rejeitas. Escreva a sub-rotina TECLADO1, que usa a técnica proposta para tratar o problema dos rebotes.

EP 3.26. Vamos propor um ambiente mais complexo para usar o teclado matricial. Cada tecla terá ação sobre os *leds*, como especificado na Tabela 3.13. Por exemplo, a tecla 1 acende o *LED1*, a tecla 2 o apaga e a tecla 3 inverte seu estado.

Tabela 3.13. Comandos do teclado matricial para acionar os *leds*.

	Tecla	Ação		Tecla	Ação		Tecla	Ação
LED1	1	Acende		2	Apaga		3	Inverte
LED2	4	Acende		5	Apaga		6	Inverte
Ambos	7	Acende		8	Apaga		9	Inverte

Usando as sub-rotinas do ER 3.17 e a fila circular sugerida no ER 2.21 (Capítulo 2) proponha a sub-rotina CMD_LEDS, para ler o teclado e comandar os *leds*, como sugerido na Figura 3.59. Neste exercício tão simples, a fila circular não é realmente necessária. A estamos usando apenas para praticar o conceito.

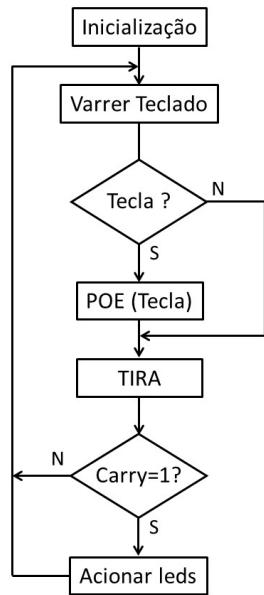


Figura 3.59. Teclado matricial usado para comandar os leds.

Gabarito para a configuração dos registradores de GPIO

	7	6	5	4	3	2	1	0
PnDIR	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
PnIN	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
PnOUT	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
PnREN	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
PnDS	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
PnSEL	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
PnIE (P1 e P2)	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
PnIES (P1 e P2)	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
PnIFG (P1 e P2)	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
PnIV (P1 e P2)	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>