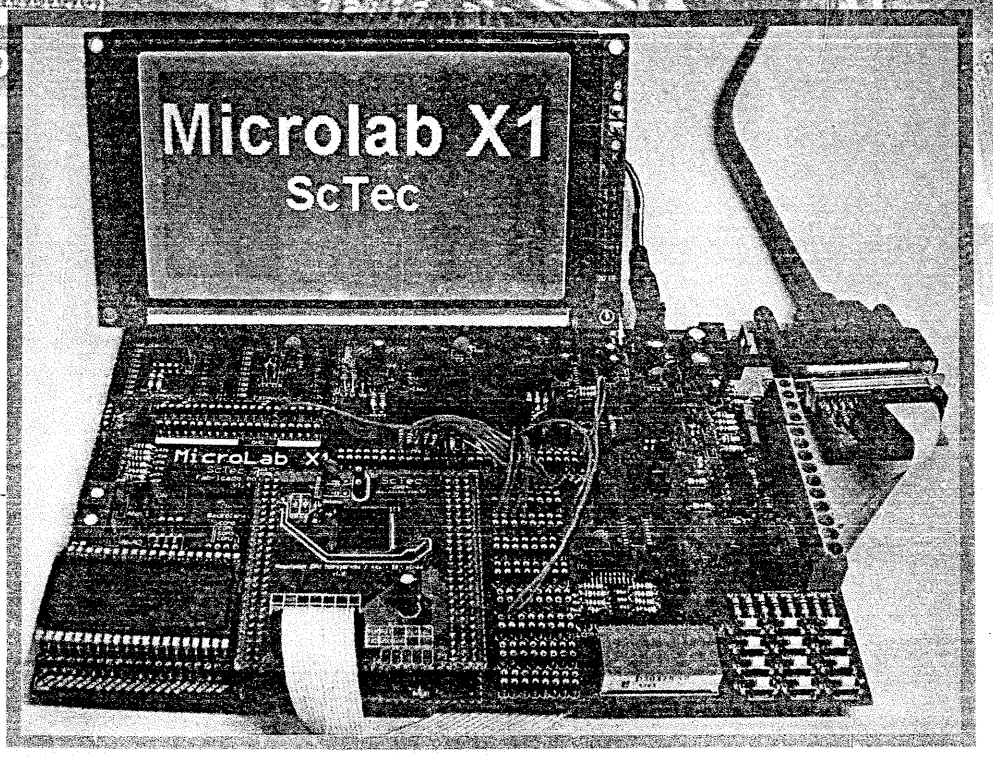
Fábio Pereira



Microcontroladores

Família MSP430

Teoria e Prática

BRINDE

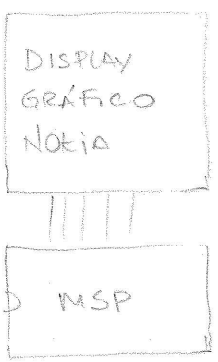
Projetos dos exemplos apresentados no capítulo 07 do livro disponíveis na Internet.

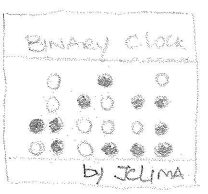
EDITORA ÉRICA



Fábio Pereira

Microcontroladores MSP430
Teoria e Prática





Editora Érica Ltda.
2005 - 1ª Edição

Conselho Editorial:

Diretor Editorial:	Antonio Marco Vicari Cipelli
Diretor Comercial:	Paulo Roberto Alves
Diretor de Publicidade:	Waldir João Sandrini
Capa e Revisão de Editoração:	Maurício Scervianinas de França
Editoração:	Érica Regina Pagano
Desenhos:	Pedro Paulo Vieira Herruzo
	Flávio Eugenio de Lima
Revisão Gramatical:	Marlene Teresa Santin Alves
Coordenação e Revisão:	Rosana Arruda da Silva

Copyright © 2005 da Editora Érica Ltda.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Pereira, Fábio, 1974 -
Microcontroladores MSP430: Teoria e Prática / Fábio Pereira. -- 1. ed. -- São Paulo: Érica, 2005.

Bibliografia.
ISBN 85-365-0067-0

1. Microcontroladores. I. Título.

05-4137 CDD-004.165

Índices para catálogo sistemático
1. Microcontroladores MSP430: Processamento de dados: Ciência da computação 004.165

Todos os direitos reservados. Proibida a reprodução total ou parcial, por qualquer meio ou processo, especialmente por sistemas gráficos, microfílmicos, fotográficos, reprográficos, fonográficos, videográficos, internet, e-books. Vedada a memorização e/ou recuperação total ou parcial em qualquer sistema de processamento de dados e a inclusão de qualquer parte da obra em qualquer programa juscibernético. Essas proibições aplicam-se também às características gráficas da obra e à sua editoração. A violação dos direitos autorais é punível como crime (art. 184 e parágrafos, do Código Penal, conforme Lei nº 10.695, de 07.01.2003) com pena de reclusão, de dois a quatro anos, e multa, conjuntamente com busca e apreensão e indenizações diversas (artigos 102, 103 parágrafo único, 104, 105, 106 e 107 itens 1, 2 e 3 da Lei nº 9.610, de 19/06/98, Lei dos Direitos Autorais).

O Autor e a Editora acreditam que todas as informações aqui apresentadas estão corretas e podem ser utilizadas para qualquer fim legal. Entretanto, não existe qualquer garantia, explícita ou implícita, de que o uso de tais informações conduzirá sempre ao resultado desejado. Os nomes de sites e empresas, porventura mencionados, foram utilizados apenas para ilustrar os exemplos, não tendo vínculo nenhum com o livro, não garantindo a sua existência nem divulgação. Eventuais erratas estarão disponíveis no site da Editora Érica para download.

Editora Érica Ltda.
Rua São Gil, 159 - Tatuapé
CEP: 03401-030 - São Paulo - SP
Fone: (11) 295-3066 - Fax: (11) 6197-4060
www.editoraerica.com.br

Requisitos de *Hardware* e de *Software*

Hardware

Microcomputador compatível com IBM-PC, processador 300MHz ou superior, mínimo de 64MB de memória, hardware de depuração/programação compatível com o FET da Texas Instruments.

Software

Windows 98SE/NT/2000/XP ou superior.

IAR Embedded Workbench 3.1.

Modem e acesso à Internet para *download* dos exemplos do livro (veja página 10).

Sobre o Autor

Nascido em São Francisco do Sul, Fábio Pereira é formado em Direito e em Eletrônica e reside em Joinville - SC.

Desde o início da década de 90 atua na área de *software* e de *hardware*. Desenvolveu diversos sistemas em plataformas como: PCs, microprocessadores Z-80, 8086, 80386, etc. e microcontroladores MSP430, HC908, PIC, Zilog e MCS51.

É sócio da ScTec, empresa que atua na área de desenvolvimento de projetos e sistemas de automação industrial e eletrônica automotiva - www.sctec.com.br.

Também leciona as disciplinas de Linguagem C e Microcontroladores no SENAI-SC e é autor dos livros Microcontroladores PIC: Técnicas Avançadas, Microcontroladores PIC: Programação em C e Microcontroladores HC908: Teoria e Prática.

Dedicatória

Aos meus pais Pedro e Cristina e à minha avó Juçá.

Cantai ao Senhor um cântico novo,
porque ele operou maravilhas.
Sua mão e seu santo braço lhe deram a vitória.

Sl 97, 1

Agradecimentos

A todas as pessoas que contribuíram direta ou indiretamente para que este livro fosse possível, em especial:

Hamilton Ignácio, Rafael de Souza e André, da Texas Instruments do Brasil;

Luiz Eduardo Sutter (Dado) pela colaboração na revisão, além da escrita e teste de código;

Egídio Schroeder (Kraft) e Renie Marquet pela ajuda com a revisão do livro;

Wellington Messias (Well) e outros participantes do fórum ASM51 pelas sugestões;

Wagner Teixeira (PICListBR).

Prefácio

A proposta deste livro é abordar de forma clara e objetiva a família MSP430 de microcontroladores da *Texas Instruments*.

Nas próximas páginas, o leitor vai encontrar as mais diversas informações sobre o funcionamento e programação desses microcontroladores, além de uma boa coleção de exemplos de funcionamento (tanto das instruções *Assembly* quanto da maioria dos periféricos internos).

O ambiente de desenvolvimento *Embedded Workbench* da IAR é também estudado com detalhes suficientes para um elevado aproveitamento das suas capacidades.

São apresentados diversos exemplos de aplicação e técnicas de *hardware*, que demonstram como utilizar esses microcontroladores em aplicações reais.

A abordagem inicial do livro também foi alterada (em relação aos outros livros do mesmo autor) de forma a tornar mais didática e eficiente a experiência da leitura e, conseqüentemente, do aprendizado.

Em suma, o livro foi planejado não apenas para o aprendizado inicial do aluno/entusiasta, mas também para ser utilizado como referência pelo programador/projetista no seu dia-a-dia.

Sobre o Material Disponível na Internet

O material disponível no site da Editora Érica (www.editoraerica.com.br) contém os projetos dos exemplos apresentados no capítulo 7 do livro.

Para utilizar os arquivos, é necessário que você possua o ambiente IAR Embedded Workbench 3.1 ou versão mais recente instalado em sua máquina.

MSP430.exe - 310 KB

Procedimentos para Download

Acesse o site da Editora Érica: www.editoraerica.com.br. A transferência do arquivo disponível pode ser feita de duas formas:

- **Por meio do módulo pesquisa**. Localize o livro desejado, digitando palavras-chave (nome do livro ou do autor). Aparecerão os dados do livro e o arquivo para download, então dê um clique sobre o arquivo executável que será transferido.
- **Por meio do botão "Download"**. Na página principal do site, clique no item "Download". Será exibido um campo, no qual devem ser digitadas palavras-chave (nome do livro ou do autor). Serão exibidos o nome do livro e o arquivo para download. Dê um clique sobre o arquivo executável que será transferido.

Procedimentos para Descompactação

Primeiro passo: após ter transferido o arquivo para sua máquina, verifique o diretório em que se encontra e dê um duplo-clique sobre o arquivo. Será exibida uma tela do programa *WinZip Self-Extractor* que o conduzirá ao processo de descompactação. Abaixo da opção *Unzip to Folder*, existe um campo que indica o destino dos arquivos que serão copiados para o disco rígido do seu computador.

C:\MSP430

Segundo passo: prossiga a instalação, clicando no botão *Unzip*, o qual se encarrega de descompactar os arquivos. Logo abaixo dessa tela, aparece a barra de *status,* a qual monitora o processo para que você acompanhe. Após o término da descompressão, outra tela de informação surgirá, indicando que os arquivos foram descompactados com sucesso e estão no diretório indicado. Para sair dessa tela, clique no botão *OK,* e para finalizar o programa *WinZip Self-Extractor*, clique no botão *Close*.

Índice Analítico

Capítulo 1. Introdução **17**

1.1. *Hardware* Utilizado 18
1.2. Convenções Adotadas 21

Capítulo 2. A Arquitetura MSP430 ..... **22**

2.1. Visão Geral da CPU 23
2.1.1. Contador de Programa (R0) 24
2.1.2. Apontador da Pilha (R1) 24
2.1.3. Registrador SR/CG1 (R2) 25
2.1.4. Registradores Geradores de Constantes (R2 e R3) 27
2.1.5. Registradores de Propósito Geral (R4 a R15) 27
2.1.6. Organização da Memória 27
2.2. Modos de Operação 28
2.3. Nomenclatura e Modelos Disponíveis 30
2.4. Encapsulamentos 32
2.5. Pinagens 32

Capítulo 3. O Ambiente *Embedded Workbench* **35**

3.1. Iniciando um Novo Projeto em *Assembly* 36
3.1.1. Montando o Programa 41
3.1.2. Simulando a Execução 42
3.2. Iniciando um Novo Projeto em C 50
3.2.1. Compilando um Projeto em C 51
3.2.2. Simulando um Projeto em C 51
3.2.3. Terminal de I/O 52

Capítulo 4. Instruções *Assembly* **53**

4.1. Construção dos *Op-codes* 53
4.1.1. Instruções com Um Operando 53
4.1.2. Instruções com Dois Operandos 54
4.1.3. Instruções de Desvio 55
4.2. Modos de Endereçamento 56
4.2.1. Modo Imediato 57
4.2.2. Modo Registrador 59
4.2.3. Modo Indexado 61
4.2.4. Modo Simbólico 62
4.2.5. Modo Absoluto 63
4.2.6. Modo Indireto 64
4.2.7. Modo Indireto com Auto-Incremento 66

4.3. Instruções Físicas e Instruções Emuladas 68
4.4. Conjunto de Instruções 70
4.4.1. Instruções de Movimentação e Manipulação de Dados 71
4.4.2. Instruções Aritméticas e Lógicas 77
4.4.3. Instruções de Teste e Desvio 93
4.4.4. Instruções de Controle do Processador 100
4.5. Temporização das Instruções 103

Capítulo 5. Periféricos e Módulos Internos 105

5.1. Sistema de *Reset* 105
5.1.1. Sistema BOR 106
5.1.2. Efeitos do *Reset* 106
5.2. Sistema de Interrupções 107
5.2.1. Categorias de Interrupção 107
5.2.2. Vetores de Interrupção 108
5.2.3. Registradores de Controle de Interrupção 111
5.2.4. Tratamento de Interrupções 112
5.3. Módulo Oscilador 116
5.3.1. Oscilador de Baixa/Alta Freqüência (LFXT1) 116
5.3.2. Oscilador de Alta Freqüência (XT2) 118
5.3.3. DCO 118
5.3.4. FLL 120
5.3.5. Sinais de *Clock* Internos (MCLK, SMCLK e ACLK) 122
5.3.6. Gerenciamento de Falha no Oscilador 124
5.3.7. Seleção das Fontes de *Clock* 126
5.3.8. Registradores do Módulo Oscilador 127
5.3.9. Exemplos de Configuração 133
5.4. Portas de E/S 138
5.4.1. Registradores das Portas de E/S 139
5.4.2. Exemplos de Configuração 141
5.5. *Timer* A 143
5.5.1. *Reset* do Contador 145
5.5.2. Modo de Captura 145
5.5.3. Modo de Comparação/PWM 146
5.5.4. Interrupções do *Timer* A 149
5.5.5. Conexões do *Timer* A 149
5.5.6. Registradores do *Timer* A 150
5.5.7. Exemplos de Utilização 154
5.6. *Timer B* 157
5.6.1. Largura Programável 158
5.6.2. *Latches* de Comparação 158
5.6.3. Agrupamento de Canais 159
5.6.4. Saídas Configuráveis para Modo de Alta Impedância 160

5.6.5. Disparo de Conversão do ADC após uma Comparação 160
5.6.6. Interrupções do *Timer* B 160
5.6.7. Conexões do *Timer* B 160
5.6.8. Registradores do *Timer* B 161
5.7. Temporizador Básico (*Timer 1*) 166
5.7.1. Contador 1 166
5.7.2. Contador 2 167
5.7.3. Interrupção do Temporizador Básico 167
5.7.4. Registradores do Temporizador Básico 167
5.7.5. Exemplo de Utilização 169
5.8. USART - Modo Assíncrono 171
5.8.1. Gerador de *Baud-Rate* 172
5.8.2. Configuração da USART 175
5.8.3. Transmissão Serial 175
5.8.4. Recepção Serial 176
5.8.5. Endereçamento *Idle-Line* 177
5.8.6. Endereçamento por *bit* 179
5.8.7. Interrupções da USART 180
5.8.8. Conexões da USART 180
5.8.9. Registradores da USART 181
5.8.10. Exemplos de Utilização 186
5.9. USART - Modo Síncrono SPI 188
5.9.1. Gerador de *Clock* 190
5.9.2. Configuração da USART 190
5.9.3. Operação no Modo Mestre 191
5.9.4. Operação no Modo Escravo 191
5.9.5. Interrupções 192
5.9.6. Conexões da USART 192
5.9.7. Registradores da USART no Modo SPI 193
5.9.8. Exemplos de Utilização 197
5.10. USART - Modo Síncrono I^2C 199
5.10.1. Características do Protocolo 199
5.10.2. Características do *Hardware* 202
5.10.3. Gerador de *Clock* 204
5.10.4. Configuração da USART 204
5.10.5. Operação no Modo Mestre 205
5.10.6. Operação no Modo Escravo 207
5.10.7. Operação com DMA 208
5.10.8. Interrupções 209
5.10.9. Conexões da USART 209
5.10.10. Registradores da USART no Modo I^2C 210
5.10.11. Exemplos de Utilização 216
5.11. Comparador Analógico 223

5.11.1. Facilidades do Comparador + ... 225
5.11.2. Interrupção do Comparador 226
5.11.3. Conexões do Comparador 226
5.11.4. Registradores do Comparador Analógico 226
5.12. Amplificador Operacional 229
5.12.1. Modos de Operação 230
5.12.2. Conexões dos Amplificadores Operacionais 236
5.12.3. Registradores do Amplificador Operacional 236
5.13. Conversor A/D *Slope* 237
5.14. Conversor A/D de 10 *Bits* 239
5.14.1. Referências de Tensão 242
5.14.2. Controlador de Transferência de Dados 243
5.14.3. Configuração 244
5.14.4. Sensor de Temperatura 244
5.14.5. Interrupções 245
5.14.6. Conexões do ADC10 245
5.14.7. Registradores 245
5.15. Conversor A/D de 12 *Bits* 251
5.15.1. Referências de Tensão 255
5.15.2. Configuração 256
5.15.3. Sensor de Temperatura 256
5.15.4. Interrupções 257
5.15.5. Conexões do ADC12 257
5.15.6. Registradores 258
5.15.7. Exemplo de Utilização 264
5.16. Conversor Digital-Analógico 265
5.16.1. Referências de Tensão 268
5.16.2. Configuração 268
5.16.3. Operação com DMA 268
5.16.4. Interrupções 268
5.16.5. Conexões do DAC12 268
5.16.6. Registradores do DAC12 269
5.16.7. Exemplo de Configuração 271
5.17. Controlador de LCD 272
5.17.1. Modo Estático 273
5.17.2. Modo 2MUX 275
5.17.3. Modo 3MUX 276
5.17.4. Modo 4MUX 277
5.17.5. Conexões do Controlador de LCD 277
5.17.6. Registradores do Controlador de LCD 278
5.17.7. Exemplo de Utilização 279
5.18. Multiplicador por *Hardware* 283
5.18.1. Registradores 284

5.18.2. Exemplos de Utilização 285
5.19. Controlador de DMA 287
5.19.1. Modos de Operação e Endereçamento 287
5.19.2. Eventos de Disparo do DMA 290
5.19.3. Encerrando uma Operação DMA 292
5.19.4. Prioridades entre os Canais 292
5.19.5. Interrupções e DMA 292
5.19.6. Características de Temporização das Transferências 293
5.19.7. Conexões do Controlador de DMA 293
5.19.8. Registradores do Controlador de DMA 293
5.19.9. Exemplos de Utilização 297
5.20. Supervisor de Tensão 300
5.20.1. Conexões do SVS 301
5.20.2. Registradores do Supervisor de Tensão 301
5.21. Controlador da Memória *FLASH* 302
5.21.1. Apagamento da Memória 302
5.21.2. Programando a *FLASH* 304
5.21.3. Encerrando Prematuramente uma Operação 305
5.21.4. Interrupções do Controlador da *FLASH* 306
5.21.5. Registradores do Controlador de Memória *FLASH* 306
5.21.6. Exemplos de Utilização 309
5.22. *Watchdog* 311
5.22.1. Modo *Watchdog* 311
5.22.2. Modo Temporizador 312
5.22.3. WDT+ 312
5.22.4. Registradores do *Watchdog* 313
5.22.5. Exemplos de Utilização 314
5.23. Pino $\overline{\text{RST}}$/NMI 315
5.23.1. Registradores de Controle 315
5.24. Interfaces de Programação/Depuração 316
5.24.1. JTAG 316
5.24.2. *Bootstrap Loader* (BSL) 319
5.24.3. Depuração pela Interface JTAG 322

Capítulo 6. Programação em C 323

6.1. Revisão da Linguagem C 323
6.1.1. Comandos e Palavras Reservadas 323
6.1.2. Estrutura Básica de um Programa 323
6.1.3. Tipos de Dados 324
6.1.4. Operadores e Expressões em C 327
6.1.5. Tipos de Dados Complexos 329
6.1.6. Comandos da Linguagem C 333
6.1.7. Funções 335

6.2. O Compilador IAR 338
6.2.1. Tipos e Organização dos Dados 338
6.2.2. Convenções de Chamada e de Retorno de Funções 341
6.2.3. Funções Intrínsecas 342
6.2.4. Diretivas e Extensões da Linguagem 345
6.2.5. Bibliotecas C 347
6.2.6. Embutindo Código Assembly 350
6.2.7. Produzindo Código C Eficiente 350

Capítulo 7. Exemplos de Aplicação 352

7.1. Controlando o DCO 352
7.2. Módulo LCD 16x2 Caracteres 356
7.2.1. Voltímetro Digital Simples 365
7.2.2. Termômetro Digital 366
7.3. Display LCD Gráfico 367
7.3.1. Voltímetro Digital Gráfico 378
7.4. Varredura de Teclado 380
7.5. Controlando um Servo com PWM 382
7.6. Comunicação Serial Assíncrona 384
7.7. Comunicação SPI 390
7.8. Técnicas de Baixo Consumo 396
7.8.1. Relógio de Baixo Consumo 397

Apêndice A - Conjunto de Instruções *Assembly* 402

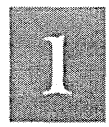
Apêndice B - Funções da Biblioteca C 404

Apêndice C - Tabela ASCII 408

Índice Remissivo 409

Referências Bibliográficas 412

Marcas Registradas 414



Introdução

Neste livro, vamos estudar os microcontroladores MSP430 da *Texas Instruments*, bem como as suas ferramentas de programação e exemplos de utilização.

No decorrer dos capítulos são abordados a arquitetura básica dos *chips*, o ambiente de desenvolvimento, conjunto de instruções *Assembly*, periféricos internos, o compilador C, além de exemplos de aplicação.

É oportuno esclarecer ao leitor que foi adotada a linguagem C.

Mas por que C em vez de *Assembly*?

Esta é uma questão bastante controversa, mas em poucas palavras:

1. C é uma linguagem de programação de alto nível (quando comparada com *Assembly*) e por isso é muito mais produtiva que *Assembly*.
2. C é uma linguagem altamente portável, ou seja, os programas escritos para um *chip* podem ser facilmente adaptados para funcionar em outro *chip*.
3. C é uma linguagem altamente eficiente. Um bom programador C pode gerar um código quase tão eficiente quanto em *Assembly*, mas em um tempo muitíssimo menor.
4. Utilizando C, a curva de aprendizado de um novo microcontrolador pode ser substancialmente reduzida, uma vez que o programador tem de se preocupar basicamente com os periféricos e não com a linguagem do *chip*.
5. Os microcontroladores MSP430 possuem uma arquitetura de *hardware* que favorece muito a utilização da linguagem C.

É claro que não estamos decretando a morte do *Assembly*, muito pelo contrário, em muitos casos, especialmente em rotinas críticas de tempo, pode ser essencial o uso do *Assembly*.

Além disso, o conhecimento da linguagem *Assembly* é muito importante no sentido de estudar o código gerado pelo compilador. Muitas vezes o exame do código gerado pode conduzir o programador a uma nova abordagem de programação C, de forma a tornar o código mais eficiente.

Por isso, a linguagem *Assembly* não será (e nem poderia ser) deixada de lado, mas fica aqui a sugestão: se você ainda não é um programador em linguagem C, a hora de começar a aprender é agora.

Este livro não é exatamente a ferramenta ideal para um leigo aprender C, mas dependendo da força de vontade do programador, ele pode ser um bom começo.

Nas referências bibliográficas o leitor pode encontrar alguns outros livros que podem servir ao aprendizado da linguagem C.

1.1. *Hardware* Utilizado

Todos os exemplos desenvolvidos para o livro foram implementados na placa Microlab X1, uma estação de desenvolvimento projetada para o uso com diversas plataformas de microcontroladores, microprocessadores e DSPs.

No presente caso, foram utilizadas placas adaptadoras para MSP430F149 e MSP430F449. A figura 1-1 apresenta a placa utilizada.

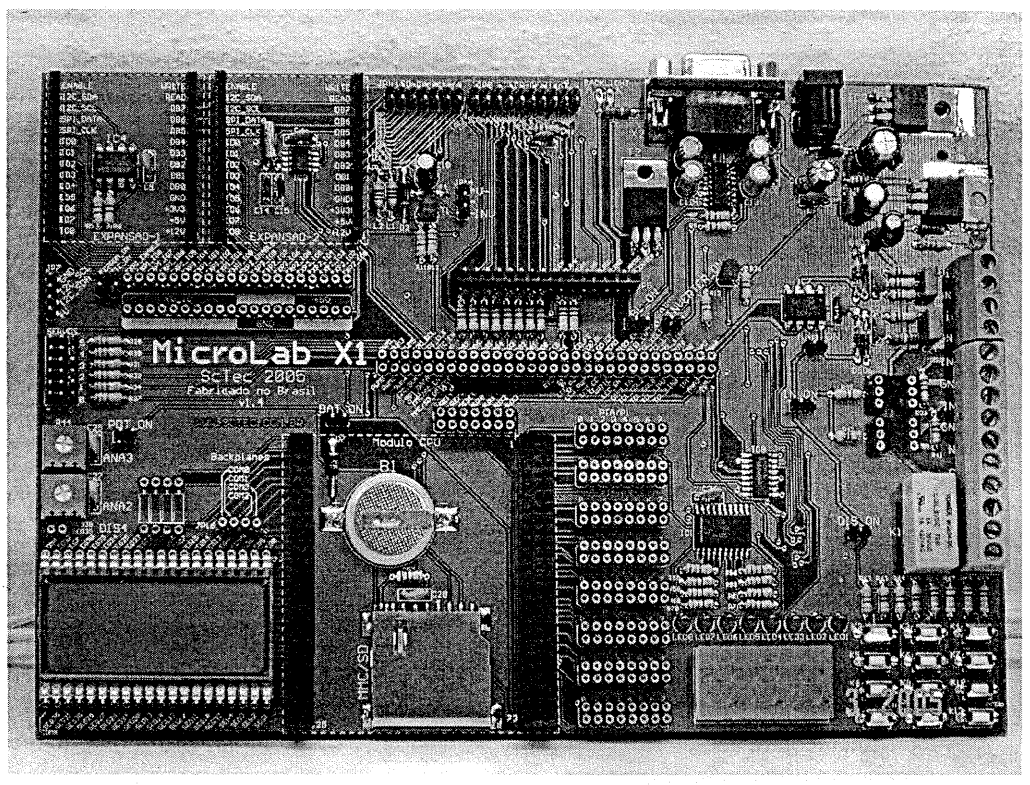


Figura 1-1

O circuito básico de ligação da CPU MSP430F149 (válido também para os *chips* 13x, 14x, 15x e 16x) pode ser visto na figura 1-2. Os capacitores C1, C2, C5 e C6 devem ser de 8,2pF, mas podem ser suprimidos, dependendo do cristal utilizado. Nos exemplos do livro, utilizamos um cristal de 32.768Hz para Q2. Q1 não foi montado.

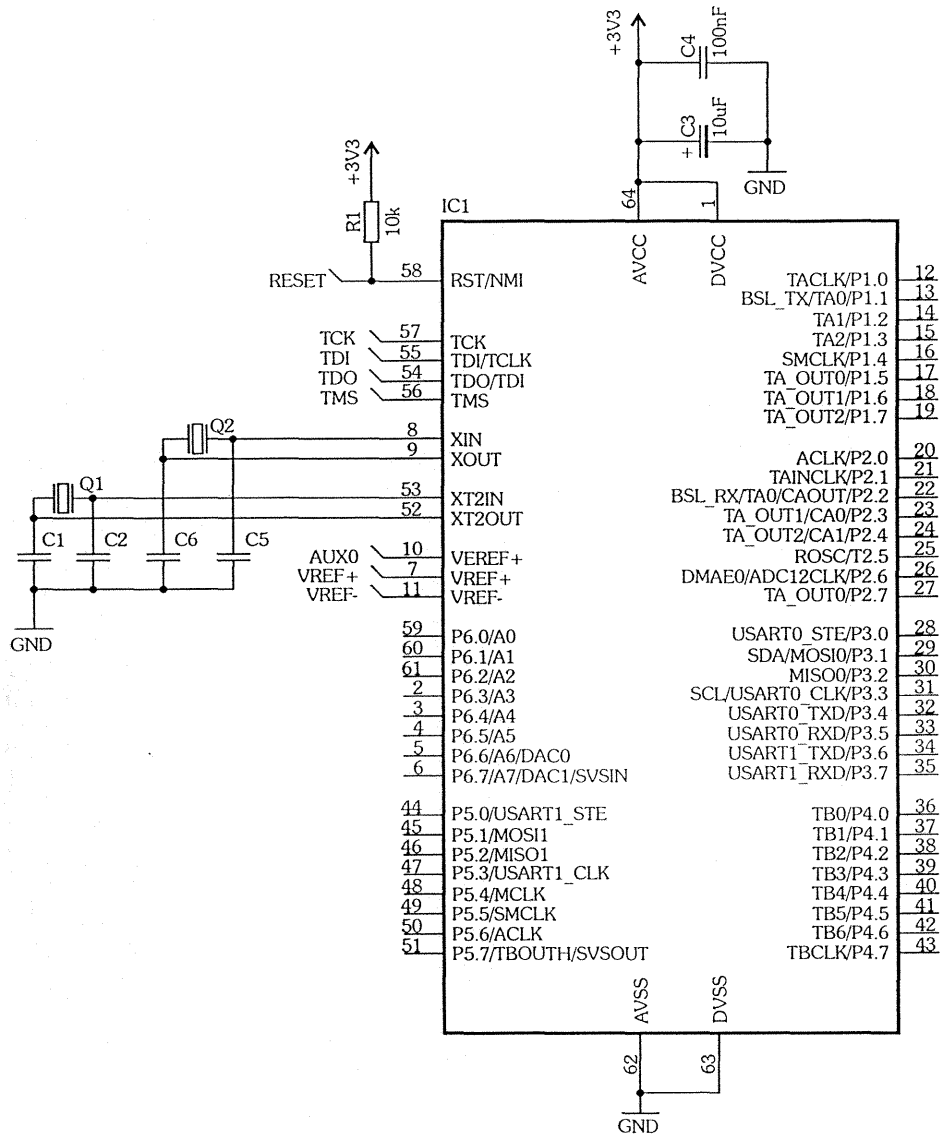


Figura 1-2

Na figura 1-3 podemos ver o circuito básico utilizado para a CPU MSP430F449. Os capacitores C1, C2, C5 e C6 devem ser de 8,2pF, mas podem ser suprimidos, dependendo do cristal utilizado.

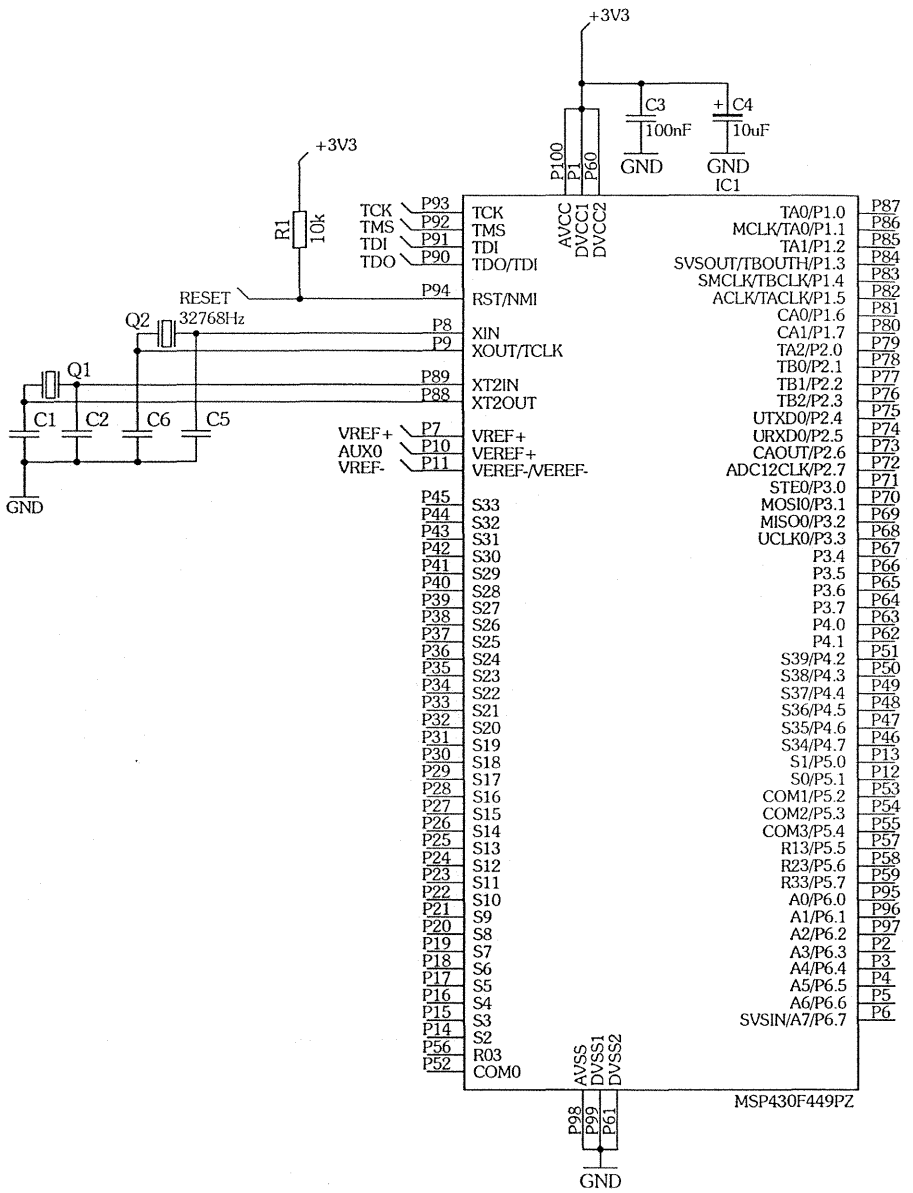


Figura 1-3

A figura 1-4 apresenta a conexão de um LED ao pino P1.0, que será utilizada em alguns exemplos no decorrer deste livro. Eventualmente, o mesmo circuito pode ser conectado a outros pinos, se for especificado no exemplo.

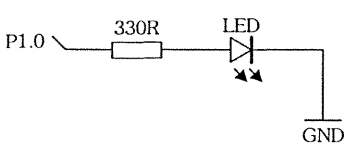


Figura 1-4

O circuito utilizado para comunicação serial com um microcomputador PC pode ser visto na figura 1-5. Observe que foi utilizado um MAX232 (ou equivalente), alimentado por uma fonte de 5V, cujo objetivo é facilitar a aquisição do componente por parte do leitor. A versão de 3,3 Volts do componente (MAX232) não é encontrada facilmente.

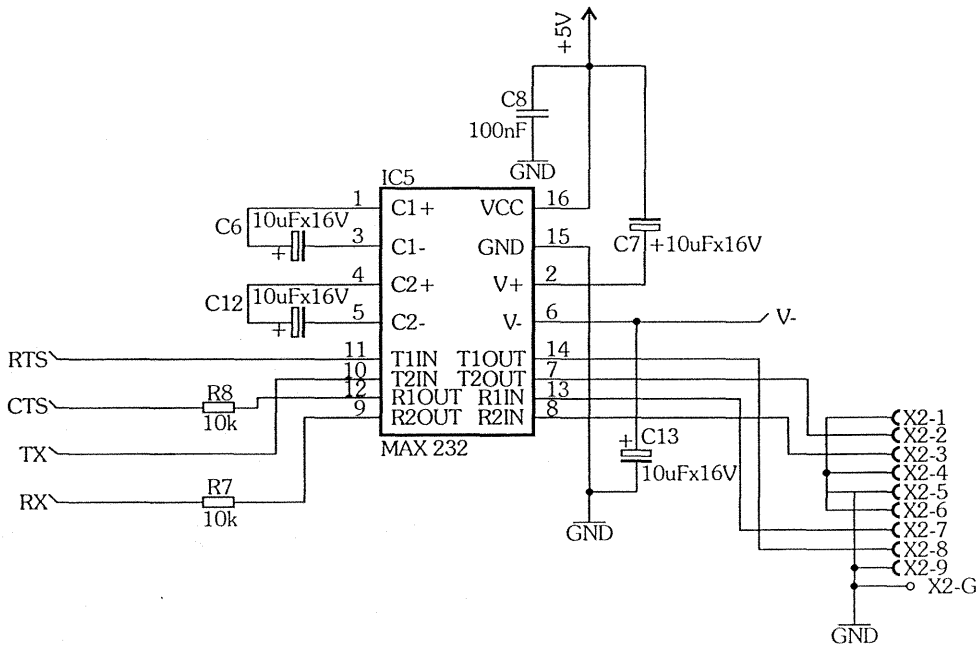


Figura 1-5

1.2. Convenções Adotadas

Antes de começar a leitura, é importante conhecer algumas convenções adotadas na escrita:

1. As palavras em idioma estrangeiro estão preferencialmente grafadas em *itálico*.
2. A citação de nomes de registradores e de *bits* é feita sempre em letras maiúsculas.
3. A referência a um *bit* específico em um registrador é feita apenas pelo seu nome ou utiliza a forma NOME DO REGISTRADOR : NOME DO BIT, assim o *bit* GIE localizado no registrador SR pode ser referenciado somente pelo seu nome GIE, ou SR:GIE.
4. A utilização da letra "x" no meio de um nome de registrador indica uma referência geral a diversos registradores pertencentes ao mesmo módulo. Por exemplo: PxOUT é uma referência geral aos registradores de saída das portas de E/S. Podemos ter registradores chamados P1OUT, P2OUT, P3OUT e assim por diante.
5. Quando existirem símbolos especiais (definidos nos arquivos de cabeçalho "io430xxx.h") para a configuração de um ou mais *bits* de um registrador, eles vão aparecer grafados em negrito, seguindo a descrição do(s) *bit*(s).
6. No texto do livro, as palavras reservadas da linguagem C sempre são grafadas com caracteres em negrito (nos programas exemplo a grafia é normal).

2

A Arquitetura MSP430

Este capítulo trata das principais características dos *chips* MSP430, especialmente o funcionamento da sua CPU, as modalidades de operação e organização da memória.

Alguns dos aspectos-chave da arquitetura MSP430 são:

- **Baixo consumo** - os MSP430 são *chips* conhecidos pelo seu consumo incrivelmente baixo (da ordem de 0,1µA para retenção dos dados na RAM, 0,8µA para funcionamento no modo de relógio de tempo real e cerca de 250 µA/MIPS em funcionamento normal). O baixo consumo é obtido graças aos diversos modos de funcionamento da CPU, conforme veremos mais adiante.
- **Baixa tensão de operação** - os MSP430 podem operar com tensões a partir de 1,8V até 3,6 Volts (a tensão mínima para programação da FLASH é 2,2V para os dispositivos da família 2xx e 2,7V para os demais).
- **Alta performance** - utilizando um barramento de dados de 16 *bits* (ao contrário da grande maioria dos seus competidores diretos, que são *chips* de 8 *bits*), diversos modos de endereçamento e um conjunto de instruções pequeno, mas muitíssimo poderoso, os MSP430 permitem realizar tarefas complexas com um código bastante pequeno e rápido.
- **Conjunto de instruções ortogonais** - a disponibilidade de qualquer modo de endereçamento para qualquer instrução e qualquer operando permite que se escrevam códigos pequenos e eficientes, facilitando a tarefa dos compiladores de linguagens de alto nível como a linguagem C.
- **Número reduzido de instruções** - arquitetura RISC com apenas 27 instruções físicas (*op-codes*) e mais 24 instruções emuladas (variações das 27 instruções que utilizam os geradores de constantes), resultando um conjunto de 51 instruções.
- **Grande quantidade de periféricos** - os *chips* MSP430 contam com um conjunto bastante extenso de periféricos internos, com uma ênfase especial para os conversores AD de até 16 *bits*, conversores DA, comparador analógico, amplificador operacional programável, controladores de DMA, *timers* com diversos modos de funcionamento (incluindo PWM), controlador de LCD, USARTs com capacidade de endereçamento, multiplicador por *hardware* com capacidade de executar operações de multiplicação e acúmulo, etc.
- **Facilidade de gravação e de depuração** - a utilização da interface JTAG (do acrônimo inglês *Joint Test Action Group*) para gravação e depuração permite que o projetista realize a programação e a depuração do seu *software* diretamente na placa de aplicação, sem a necessidade de utilização de equipamentos dispendiosos como emuladores.
- **Diversos encapsulamentos** - desde o diminuto QFN de 24 pinos e seus 4 x 4 mm até encapsulamentos LQFP de 100 pinos. Os dispositivos das famílias 1xx, 2xx e 4xx não possuem versões com encapsulamento DIP.

Vejamos agora alguns detalhes internos do funcionamento da CPU.

2.1. Visão Geral da CPU

Os microcontroladores MSP430 possuem um design simples e ao mesmo tempo poderoso.

Sua arquitetura RISC combina um conjunto reduzido de instruções (apesar de uma parcela dos especialistas em arquitetura de processadores discordar de que sejam *chips* RISC, pelo fato de possuírem instruções com larguras e, conseqüentemente, tempos de execução variáveis) com uma arquitetura de barramento clássica Von Neumann, permitindo que a CPU possua um espaço único de endereçamento de memória.

Desta forma, em tese, não há distinção entre memória de programa e memória de dados. Claro que, funcionalmente, esta afirmação não é verdadeira, já que alguns endereços são populados por registradores de acesso a periféricos, outros são utilizados para RAM de uso geral, enquanto outros endereços são preenchidos com memórias não-voláteis do tipo ROM, PROM ou FLASH, mais adequadas ao armazenamento do programa do usuário.

Em primeiro lugar, é importante saber que a CPU desses *chips* possui três barramentos distintos (endereços, dados e controle), sendo os dois primeiros de 16 *bits*.

Uma vez que o barramento de endereços possui largura de 16 *bits*, significa que podemos acessar até 65.536 posições de memória.

Já com relação ao barramento de dados, a largura de 16 *bits* significa que a CPU pode processar informações em lotes de 16 *bits* (a maioria dos concorrentes diretos dos MSP430 são *chips* de 8 *bits*). Isso facilita muito o trabalho do programador, pois muitas aplicações atuais trabalham com dados de 16 ou mais *bits*.

Além disso, a CPU possui também 16 registradores internos (todos de 16 *bits*), nomeados de R0 a R15.

Registrador	Nome/Função
R0	Contador de programa (PC)
R1	Apontador da pilha (SP)
R2	*Status*/Gerador de constantes 1
R3	Gerador de constantes 2
R4	Registrador de propósito geral (GPR)
R5	Registrador de propósito geral (GPR)
R6	Registrador de propósito geral (GPR)
R7	Registrador de propósito geral (GPR)
R8	Registrador de propósito geral (GPR)
R9	Registrador de propósito geral (GPR)
R10	Registrador de propósito geral (GPR)
R11	Registrador de propósito geral (GPR)
R12	Registrador de propósito geral (GPR)
R13	Registrador de propósito geral (GPR)
R14	Registrador de propósito geral (GPR)
R15	Registrador de propósito geral (GPR)

Tabela 2-1

Os quatro primeiros registradores possuem finalidades dedicadas (especificadas pelo fabricante) e via de regra não podem ser utilizados para outros propósitos.

Os demais registradores (R4 a R15) podem ser utilizados para propósitos gerais (como o armazenamento de variáveis do usuário, apontadores, etc.).

Repare que os MSP430 não possuem um registrador acumulador específico: qualquer um dos 16 registradores da CPU pode funcionar como fonte ou destino de uma operação. Além disso, qualquer dos registradores da CPU e qualquer endereço da memória pode funcionar como fonte e/ou destino para uma operação. Isso garante uma enorme flexibilidade na escrita de programas.

Os MSP430 permitem ainda que se realizem operações envolvendo operandos de 8 ou 16 *bits*.

Numa operação de escrita de 8 *bits*, tendo um registrador da CPU como destino, o *byte* mais significativo do registrador é preenchido com o valor zero, ao passo que numa operação de leitura de 8 *bits*, tendo um registrador da CPU como fonte, somente os 8 *bits* menos significativos são considerados.

2.1.1. Contador de Programa (R0)

O contador de programa (PC - ***Program Counter***) possui a finalidade de apontar a próxima instrução a ser lida da memória e executada pela CPU.

Como se trata de um registrador de 16 *bits*, deduz-se que o espaço total de endereçamento dos MSP430 é de 64K ou 65.536 endereços. Esse espaço de endereçamento é organizado em *bytes*, estando as instruções localizadas sempre nos endereços pares da memória (todas as instruções ocupam, no mínimo, 16 *bits*). Isso implica em que o *bit* 0 desse registrador é sempre mantido em nível lógico "0".

Um aspecto interessante do contador de programa nessa arquitetura é que ele pode ser lido/escrito diretamente pelo *software* em execução, permitindo o uso de técnicas, como, por exemplo, o desvio calculado.

Após o *reset* do sistema, o registrador PC é carregado com o conteúdo do vetor de *reset*, especificado no endereço 0xFFFE. O endereço apontado por ele corresponde à localização da primeira instrução do programa do usuário.

2.1.2. Apontador da Pilha (R1)

O apontador da pilha (SP - ***Stack Pointer***) é utilizado para indicar à CPU a localização do topo da pilha de memória.

A pilha de memória, ou simplesmente pilha, é utilizada para o armazenamento de endereços de retorno nas chamadas de sub-rotinas e tratamento de interrupções.

A cada operação de empilhamento (armazenamento de um dado na pilha) o SP é decrementado de 2 e a cada operação de desempilhamento (leitura de um dado da pilha) o SP é incrementado de 2.

Nos MSP430 é possível também o armazenamento de outros valores na pilha por meio das instruções PUSH (armazenamento na pilha) e POP (leitura de um valor da pilha). Isso significa que é possível utilizar a pilha para o armazenamento temporário de informações, como, por

exemplo, o salvamento de contexto em interrupções, passagem de parâmetros na chamada de funções e sub-rotinas, etc.

As figuras 2-1 a 2-4 demonstram o funcionamento e utilização da pilha. As linhas em negrito indicam o topo da pilha. Na figura 2-1 temos a pilha após a inicialização do SP com o valor 1024 (0x0400). Na figura 2-2, temos o estado da pilha após uma chamada de sub-rotina ou função. Repare que o endereço de retorno (o endereço da instrução seguinte a que efetuou o desvio) é salvo na pilha e o PC é carregado com o endereço da primeira instrução da sub-rotina ou função. Após o empilhamento do endereço do retorno, o SP é decrementado em dois, de forma a apontar para a próxima posição livre da pilha.

A figura 2-3 representa o estado da pilha após o salvamento do estado do registrador SR. Neste caso, a posição imediatamente anterior ainda contém o endereço de retorno da sub-rotina ou função. A operação de empilhamento é realizada pela instrução PUSH.

Na figura 2-4, podemos observar o estado da pilha após a execução da sub-rotina ou função hipoteticamente executada no exemplo corrente.

SP	Conteúdo
1024	**?**
1022	?
1020	?
1018	?

Figura 2-1

SP	Conteúdo
1024	PC (retorno)
1022	**?**
1020	?
1018	?

Figura 2-2

SP	Conteúdo
1024	PC (retorno)
1022	SR
1020	**?**
1018	?

Figura 2-3

SP	Conteúdo
1024	**PC (retorno)**
1022	SR
1020	?
1018	?

Figura 2-4

Lembre-se de que a pilha é uma estrutura LIFO (*Last In First Out*, ou seja, o último a entrar é o primeiro a sair). Sendo assim, os dados armazenados devem ser recuperados na ordem inversa em que foram guardados.

Observe que o SP também pode funcionar como operando de quaisquer instruções do MSP430.

Estas características garantem uma enorme flexibilidade na manipulação da pilha pelo *software* do usuário.

Assim como no caso do PC, o registrador SP também possui o seu *bit* menos significativo em nível lógico "0", de forma que somente pode apontar para endereços pares de memória.

Após o *reset*, o seu conteúdo é indeterminado e deve ser inicializado pelo usuário antes que utilize a pilha.

2.1.3. Registrador SR/CG1 (R2)

O registrador R2 acumula duas funções diferentes: pode funcionar como o registrador de estado da CPU, ou ainda como um gerador de constantes.

O registrador de estado (SR) possui o propósito de armazenar *bits* de estado (*flags*) e de controle da CPU.

Bit	15 ... 9	8	7	6	5	4	3	2	1	0
	Reservados	V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C
Reset:	0 ... 0	0	0	0	0	0	0	0	0	0

O significado de cada um dos *bits* é descrito em seguida:

V - *Flag* de estouro (*overflow*). Esse *bit* indica se o resultado de uma operação envolvendo operandos sinalizados ultrapassou o limite de representação da variável.

No caso de uma operação envolvendo valores de 8 *bits* sinalizados, V será setado se o resultado for maior que +127 ou menor que –128.

No caso de uma operação envolvendo valores de 16 *bits* sinalizados, V será setado se o resultado for maior que +32767 ou menor que –32768.

GIE - *Bit* de controle global de interrupções.

0 - Interrupções mascaráveis desabilitadas;

1 - Interrupções mascaráveis habilitadas;

A descrição e o funcionamento das interrupções serão vistos mais adiante neste capítulo.

N - *Flag* de resultado negativo. Esse *bit* reflete o estado do *bit* mais significativo do resultado da operação (*bit* 15 ou *bit* 7, conforme o tipo de operação).

0 - Resultado positivo (*bit* mais significativo igual a zero);

1 - Resultado negativo (*bit* mais significativo igual a um).

Z - *Flag* de zero. Utilizado para indicar se a última operação resultou ou não em um valor igual a zero.

0 - Resultado diferente de zero;

1 - Resultado igual a zero.

C - *Flag* de transporte. Utilizado para indicar se a última operação produziu um transporte ou não.

0 - Não houve transporte;

1 - Houve transporte.

SCG1 - Gerador de *clock* do sistema. Para maiores informações sobre os *clocks* do sistema veja o tópico 5.3.

0 - Sinal SMCLK ativo;

1 - Sinal SMCLK inativo.

SCG0 - Gerador DC do DCO do sistema. Para maiores informações sobre o funcionamento do DCO, veja o tópico 5.3.3.

0 - Gerador ativo;

1 - Gerador inativo (desde que o DCO não esteja sendo utilizado para o MCLK ou SMCLK).

OSCOFF - Oscilador desligado.

0 - Oscilador LFXT1 (baixa/alta freqüência) ativo;

1 - Oscilador LFXT1 inativo (desde que não esteja sendo utilizado para o MCLK ou SMCLK).

CPUOFF - CPU desligada.

0 - CPU ativa;

1 - CPU inativa.

2.1.4. Registradores Geradores de Constantes (R2 e R3)

Os registradores R2 e R3, também chamados CG1 e CG2, possuem uma função muito especial e importante na arquitetura MSP430. Eles são responsáveis pela geração de constantes numéricas necessárias à emulação de instruções, que consiste numa extensão do conjunto físico de instruções disponível ao programador.

Na função de gerador de constantes, o registrador R2 pode assumir os valores 0x0004 ou 0x0008, enquanto o registrador R3 pode assumir os valores 0, 1, 2 ou 0xFFFF (equivalente a –1 em 16 *bits* sinalizados).

O funcionamento dos geradores de constantes e a sua importância na emulação de instruções serão vistos quando estudarmos o conjunto de instruções dos MSP430.

2.1.5. Registradores de Propósito Geral (R4 a R15)

Os registradores R4 a R15 são denominados GPRs (registradores de propósito geral) e podem ser utilizados para funções diversas à escolha do usuário, tais como: armazenamento de variáveis de uso intensivo, apontadores de endereço (também chamados de ponteiros), etc. Os registradores GPR também podem ser utilizados em operações de 8 ou 16 *bits*.

O conteúdo dos GPRs após um *reset* é indeterminado.

2.1.6. Organização da Memória

Como já dissemos, o espaço total de endereçamento dos MSP430 é de 64 *Kbytes*. A tabela 2-2 apresenta de forma resumida o mapa de memória desses *chips*:

Endereço	Descrição
0x0000 a 0x000F	SFR - Registradores de funções especiais (controle de interrupções e de ativação dos módulos internos)
0x0010 a 0x00FF	Registradores de controle de periféricos (acesso de 8 *bits*)
0x0100 a 0x01FF	Registradores de controle de periféricos (acesso de 16 *bits*)
0x0200 a 0x09FF	Memória RAM (até 2 *Kbytes*). Nos *chips* com mais de 2 *Kbytes* de RAM, essa área é espelhada nos endereços 0x1100 a 0x18FF
0x0A00 a 0x0BFF	Área não implementada
0x0C00 a 0x0FFF	ROM de *BOOT* (*Bootstrap loader*)
0x1000 a 0x10FF	*FLASH* (256 *bytes* - *Information Memory*)
0x1100 a 0x38FF	Memória RAM (até 8 *Kbytes*) ou *FLASH*, dependendo do modelo do *chip*
0x3900 a 0xFFDF	Memória *FLASH*
0xFFE0 a 0xFFFD	15 vetores de interrupção
0xFFFE a 0xFFFF	Vetor de *reset*

Tabela 2-2

Observe que a quantidade de memórias RAM e *FLASH* implementadas varia conforme o modelo do *chip* utilizado.

Cada posição de memória é formada por um *byte* e a CPU pode endereçar *bytes* individuais ou palavras (16 *bits*) individuais. Neste caso, as palavras são alinhadas nos endereços pares, com o *byte* menos significativo armazenado nos endereços pares e o *byte* mais significativo armazenado nos endereços ímpares.

Nas operações de 8 *bits*, podemos acessar tanto endereços pares como ímpares, no entanto nas operações de 16 *bits*, devemos ter em mente que é possível acessar apenas endereços pares da memória!

Um aspecto interessante do mapa de memória básico da arquitetura MSP430 reside na separação existente entre as diferentes regiões da memória. Os registradores de funções especiais (SFRs) relacionados ao suporte da CPU (como, por exemplo, o controle de interrupções) estão sempre localizados nos primeiros 16 *bytes* da memória, ao passo que os registradores de periféricos de 8 *bits* e de 16 *bits* possuem regiões distintas e fixas no mapa de memória da arquitetura.

Há ainda uma área de 1.024 *bytes* de memória ROM, na qual é implementado o programa responsável pelo *Bootstrap Loader*, ou simplesmente BSL e que é responsável pela programação do *chip* por meio de uma interface serial assíncrona (maiores detalhes sobre o funcionamento do BSL serão vistos no tópico 5.24.2).

Os últimos 32 *bytes* da memória *FLASH* são ocupados pelos vetores de interrupção e de *reset*. Os vetores armazenam o endereço de memória para onde o programa deve ser desviado na ocorrência de um determinado evento.

2.2. Modos de Operação

A arquitetura dos MSP430 disponibiliza diversos modos de funcionamento que permitem um controle bastante preciso do consumo de corrente pelo *chip*.

Existem ao todo seis modos de operação disponíveis, controlados de acordo com o estado dos *bits* CPUOFF, OSCOFF, SCG0 e SCG1 do registrador SR, conforme a tabela 2-3.

A transição do modo normal para um dos modos de baixo consumo (LPM) é feita simplesmente setando/ressetando os *bits* supracitados do registrador SR. O modo selecionado é ativado logo após a operação de escrita no registrador SR.

Uma vez que o *chip* tenha entrado em um dos modos de baixo consumo (LPM0 a LPM4), a CPU vai parar a execução do programa, no entanto o estado dos registradores, pinos de E/S e conteúdo da memória RAM são preservados.

A ocorrência de uma interrupção que esteja internamente habilitada fará com que o *chip* saia do modo de baixo consumo, retornando ao modo de funcionamento normal. A seqüência desencadeada dentro do *chip* é a seguinte: primeiramente ocorre o salvamento automático dos registradores PC e SR na pilha. Em seguida, os *bits* CPUOFF, OSCOFF e SCG1 (registrador SR) são automaticamente zerados, fazendo com que a CPU seja ativada no modo normal.

Observe que, ao término da RTI (**R**otina de **T**ratamento de **I**nterrupção), o conteúdo dos registradores PC e SR é automaticamente restaurado da pilha, fazendo com que a CPU retorne ao modo em que se encontrava antes da interrupção.

Consumo	Modo	CPUOFF	OSCOFF	SCG0	SCG1	MCLK	SMCLK	ACLK	Descrição
Maior	Normal	0	0	0	0	S	S	S	Funcionamento normal, CPU ativa e todos os sinais de *clock* ativos
	LPM0	1	0	0	0	N	S	S	CPU parada e o sinal de *clock* principal (MCLK) é desativado. Os sinais de *clock* auxiliares (SMCLK e ACLK) permanecem ativos
	LPM1	1	0	1	0	N	S	S	Idem ao LPM0, mas o DCO é desativado. O gerador DC do DCO é desativado caso não esteja sendo utilizado para gerar o SMCLK ou o ACLK
	LPM2	1	0	0	1	N	N	S	Idem ao LPM1, mas o sinal SMCLK é desativado
	LPM3	1	0	1	1	N	N	S	Idem ao LPM2, mas o gerador DC do DCO é desativado
Menor	LPM4	1	1	1	1	N	N	N	A CPU e todos os sinais de *clock* são desativados

Tabela 2-3

Se for desejado que a CPU retorne da interrupção e permaneça ativa (modo normal), será necessário manipular o conteúdo do SR salvo na pilha, antes de retornar da interrupção.

A seguir temos alguns exemplos de como se realizam a entrada e a saída de modos de baixo consumo em linguagem *Assembly* e em C:

```
; seta o GIE e o CPUOFF no SR, entra no modo LPM0
BIS  #GIE+CPUOFF,SR
; Altera o valor do SR salvo na pilha, de forma que os bits CPUOFF,
; OSCOFF, SCG0 e SCG1 sejam apagados. fazendo com que o chip retorne ao
; modo normal
BIC  #CPUOFF+OSCOFF+SCG0+SCG1,0(SP)     ; modifica o SR salvo na pilha
RETI                                    ; retorna da interrupção
```

Em C, podemos utilizar as funções intrínsecas **__low_power_mode_n()** e **__low_power_mode_off_on_exit()**, disponíveis por meio da inclusão do arquivo "intrinsics.h" :

```
//entrada no modo LPM1
__low_power_mode_1();

//saída do modo de baixa potência
__low_power_mode_off_on_exit();
```

Também estão disponíveis funções macros especiais para entrada e saída dos modos de baixa potência por meio da inclusão do arquivo "io430xxxx.h":

```
//entrada no modo LPM1
LPM1;
//saída do modo LPM1
LPM1_EXIT;
```

Repare que a função para a saída do modo de baixo consumo atua da mesma forma que em *Assembly*, ou seja, modificando o conteúdo do SR salvo na pilha. Assim, essa função deve ser chamada sempre o mais próximo possível do término ou retorno da rotina de tratamento de interrupção.

2.3. Nomenclatura e Modelos Disponíveis

A linha MSP430 é composta de diversos membros divididos em famílias, de acordo com parâmetros como o tipo de memória de programa e aplicação.

Atualmente existem três famílias disponíveis e duas planejadas:

- **MSP430x1xx** - dispositivos dotados de memória de programa PROM ou ROM (MSP430C1xx) ou *FLASH* (MSP430F1xx). São dispositivos de uso geral, dotados de uma grande variedade de periféricos.
- **MSP430x2xx** - a família 2xx, prevista para o final de 2005, ampliará as fronteiras da família 1xx, com velocidades de até 16MIPS, menor consumo e novos periféricos.
- **MSP430C3xx** - são *chips* dotados de memória de programa do tipo PROM ou ROM e incluem periféricos básicos, além de um controlador de LCD interno. Esta foi a primeira família MSP disponível e sua principal aplicação é em equipamentos de medição. Nenhum novo dispositivo será lançado para essa família, que não será abordada neste livro.
- **MSP430x4xx** - dispositivos dotados de memória de programa PROM ou ROM (MSP430C4xx) ou FLASH (MSP430F4xx). São *chips* voltados principalmente para o mercado de medidores, pois incluem uma grande quantidade de periféricos, além de um controlador de LCD interno.
- **MSP430x5xx** - a família 5xx, prevista para ser lançada no ano de 2006, ampliará a velocidade da CPU para até 25MIPS, além de maior quantidade de memória e novos periféricos.

Modelo MSP430F	Flash (KB)	RAM (*Bytes*)	E/S	Timers/ CCP	Comp	ADC/DAC	Serial	LCD (seg.)	Outros	Encapsulamento
1101A	1	128	14	1/3	1	slope/-	-	-	-	A,B,C,D
1111A	2	128	14	1/3	1	slope/-	-	-	-	A,B,C,D
1121A	4	256	14	1/3	1	slope/-	-	-	-	A,B,C,D
1122	4	256	14	1/3	-	5 x 10 *bits*/-	-	-	BOR e TEMP	A,B,G
1132	8	256	14	1/3	-	5 x 10 *bits*/-	-	-	BOR e TEMP	A,B,G
122	4	256	22	1/3	1	slope/-	1 USART	-	-	E,F,G
123	8	256	22	1/3	1	slope/-	1 USART	-	-	E,F,G
1222	4	256	22	1/3	-	8 x 10 *bits*/-	1 USART	-	BOR e TEMP	E,F,G
1232	8	256	22	1/3	-	8 x 10 *bits*/-	1 USART	-	BOR e TEMP	E,F,G
133	8	256	48	2/6	1	8 x 12 *bits*/-	1 USART	-	TEMP	H,I,J
135	16	512	48	2/6	1	8 x 12 *bits*/-	1 USART	-	TEMP	H,I,J
147	32	1024	48	2/7	1	8 x 12 *bits*/-	2 USART	-	TEMP, MPY	H,I,J
1471	32	1024	48	2/7	1	slope/-	2 USART	-	MPY	H,I
148	48	2048	48	2/7	1	8 x 12 *bits*/-	2 USART	-	TEMP, MPY	H,I,J
1481	48	2048	48	2/7	1	slope/-	2 USART	-	MPY	H,I
149	60	2048	48	2/7	1	8 x 12 *bits*/-	2 USART	-	TEMP, MPY	H,I,J
1491	60	2048	48	2/7	1	slope/-	2 USART	-	MPY	H,I
155	16	512	48	2/6	1	8 x 12 *bits*/ 2 x DAC12	1 USART 1 I2C	-	BOR, SVS, TEMP, DMA	I
156	24	1024	48	2/6	1	8 x 12 *bits*/ 2 x DAC12	1 USART 1 I2C	-	BOR, SVS, TEMP, DMA	I
157	32	1024	48	2/6	1	8 x 12 *bits*/ 2 x DAC12	1 USART 1 I2C	-	BOR, SVS, TEMP, DMA	I
167	32	1024	48	2/7	1	8 x 12 *bits*/ 2 x DAC12	2 USART 1 I2C	-	BOR, SVS, TEMP, MPY, DMA	I
168	48	2048	48	2/7	1	8 x 12 *bits*/ 2 x DAC12	2 USART 1 I2C	-	BOR, TEMP, MPY, DMA	I
169	60	2048	48	2/7	1	8 x 12 *bits*/ 2 x DAC12	2 USART 1 I2C	-	BOR, SVS, TEMP, MPY, DMA	I
1610	32	5120	48	2/7	1	8 x 12 *bits*/ 2 x DAC12	2 USART 1 I2C	-	BOR, SVS, TEMP, MPY, DMA	I

Modelo MSP430F	Flash (KB)	RAM (*Bytes*)	E/S	Timers/ CCP	Comp	ADC/DAC	Serial	LCD (seg.)	Outros	Encapsulamento
1611	48	10240	48	2/7	1	8 x 12 *bits*/ 2 x DAC12	2 USART 1 I2C	-	BOR, SVS, TEMP, MPY, DMA	I
1612	55	5120	48	2/7	1	8 x 12 *bits*/ 2 x DAC12	2 USART 1 I2C	-	BOR, SVS, TEMP, MPY, DMA	I
2101	1	128	16	1/3	1	slope/-	-	-	BOR, SVS	A,B,C,D
2111	2	128	16	1/3	1	slope/-	-	-	BOR, SVS	A,B,C,D
2121	4	256	16	1/3	1	slope/-	-	-	BOR, SVS	A,B,C,D
2131	8	256	16	1/3	1	slope/-	-	-	BOR, SVS	A,B,C,D
412	4	256	48	2(1 8bit)/3	1	slope/-	-	96	BOR, SVS	H,I
413	8	256	48	2(1 8bit)/3	1	slope/-	-	96	BOR, SVS	H,I
E412	4	256	48	2(1 8bit)/3	-	2 x SD16/-	-	128	BOR, SVS, MPY, TEMP	I
E413	8	256	48	2(1 8bit)/3	-	2 x SD16/-	-	128	BOR, SVS, MPY, TEMP	I
415	16	512	48	2(1 8bit)/5	1	slope/-	-	96	BOR, SVS	I
417	32	1024	48	2(1 8bit)/5	1	slope/-	-	96	BOR, SVS	I
W423	8	256	48	2(1 8bit)/5	1	slope/-	-	96	BOR, SVS, FLOW	I
W425	16	512	48	2(1 8bit)/5	1	slope/-	-	96	BOR, SVS, FLOW	I
W427	32	1024	48	2(1 8bit)/5	1	slope/-	-	96	BOR, SVS, FLOW	I
423	8	256	14	2(1 8bit)/3	-	3 x SD16/-	1 USART	128	BOR, SVS, TEMP	I
425	16	512	14	2(1 8bit)/3	-	3 x SD16/-	1 USART	128	BOR, SVS, TEMP	I
427	32	1024	14	2(1 8bit)/3	-	3 x SD16/-	1 USART	128	BOR, SVS, TEMP	I
E423	8	256	14	2(1 8bit)/3	-	3 x SD16/-	1 USART	128	BOR, SVS, TEMP, EMETER	I
E425	16	512	14	2(1 8bit)/3	-	3 x SD16/-	1 USART	128	BOR, SVS, TEMP, EMETER	I
E427	32	1024	14	2(1 8bit)/3	-	3 x SD16/-	1 USART	128	BOR, SVS, TEMP, EMETER	I
4250	16	256	32	2(1 8bit)/3	-	SD16	-	56	BOR, TEMP, DAC12	48 pinos SOP
4260	24	256	32	2(1 8bit)/3	-	SD16	-	56	BOR, TEMP, DAC12	48 pinos SOP
4270	32	256	32	2(1 8bit)/3	-	SD16	-	56	BOR, TEMP, DAC12	48 pinos SOP
435	16	512	48	3(1 8bit)/6	1	8 x 12 *bits*/-	1 USART	128/160	BOR, SVS, TEMP	K,L
436	24	1024	48	3(1 8bit)/6	1	8 x 12 *bits*/-	1 USART	128/160	BOR, SVS, TEMP	K,L
437	32	1024	48	3(1 8bit)/6	1	8 x 12 *bits*/-	1 USART	128/160	BOR, SVS, TEMP	K,L
G437	32	1024	48	3(1 8bit)/6	1	12 x 12 *bits*/ 2 x DAC12	1 USART	128	BOR, SVS, TEMP, DMA, 3 AO	K
G438	48	2048	48	3(1 8bit)/6	1	12 x 12 *bits*/ 2 x DAC12	1 USART	128	BOR, SVS, TEMP, DMA, 3 AO	K
G439	60	2048	48	3(1 8bit)/6	1	12 x 12 *bits*/ 2 x DAC12	1 USART	128	BOR, SVS, TEMP, DMA, 3 AO	K
447	32	1024	48	3(1 8bit)/7	1	8 x 12 *bits*/-	2 USART	160	BOR, SVS, TEMP, MPY	L
448	48	1024	48	3(1 8bit)/7	1	8 x 12 *bits*/-	2 USART	160	BOR, SVS, TEMP, MPY	L
449	60	2048	48	3(1 8bit)/7	1	8 x 12 *bits*/-	2 USART	160	BOR, SVS, TEMP, MPY	L

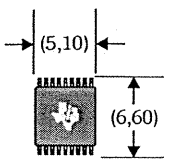
Tabela 2-4

Na tabela 2-4 foram utilizadas as seguintes abreviaturas e siglas:

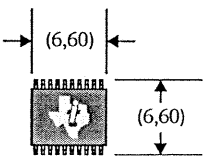
- **ADC** - conversor analógico-digital;
- **AO** - amplificador operacional;
- **BOR** - módulo de *reset* por queda de tensão (*brownout reset*);
- **Comp** - comparador analógico;
- **CCP** - módulo de captura/comparação/PWM;
- **DAC** - conversor digital-analógico;
- **DMA** - módulo de acesso direto à memória;
- **EMETER** - módulo de medição elétrica;
- **FLOW** - medidor de fluxo;
- **MPY** - multiplicação por *hardware*;
- **SVS** - supervisão de tensão;
- **TEMP** - sensor de temperatura.

2.4. Encapsulamentos

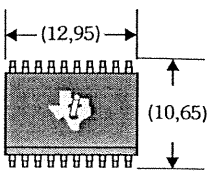
A - TVSOP 20 pinos



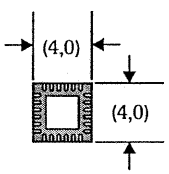
B - TSSOP 20 pinos



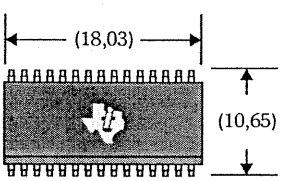
C - SOP (SOIC) 20 pinos



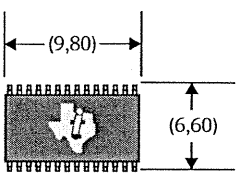
D - QFN 24 pinos



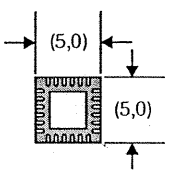
E - SOP (SOIC) 28 pinos



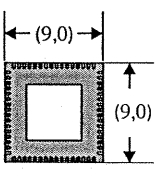
F - TSSOP 28 pinos



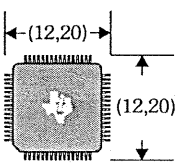
G - QFN 32 pinos



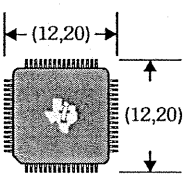
H - QFN 64 pinos



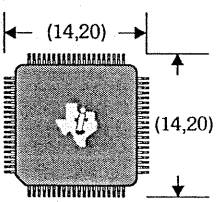
I - LQFP 64 pinos



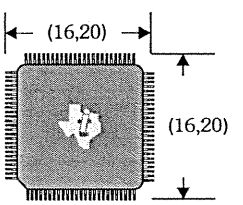
J - TQFP 64 pinos



K - LQFP 80 pinos



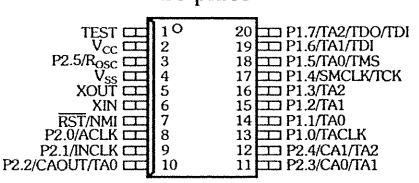
L - LQFP 100 pinos



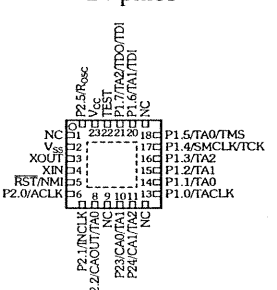
* medidas em mm.

2.5. Pinagens

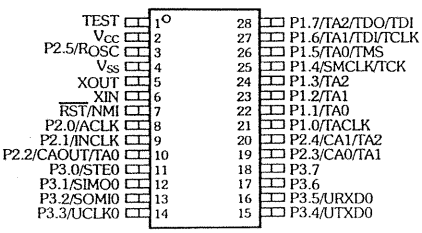
20 pinos



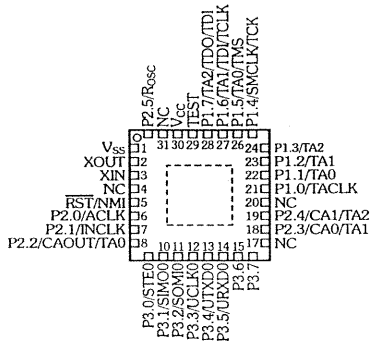
24 pinos



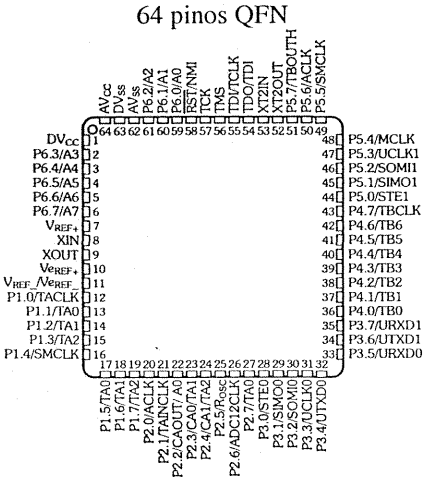
28 pinos



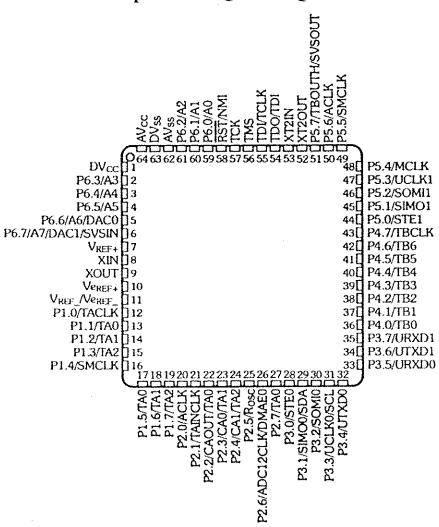
32 pinos



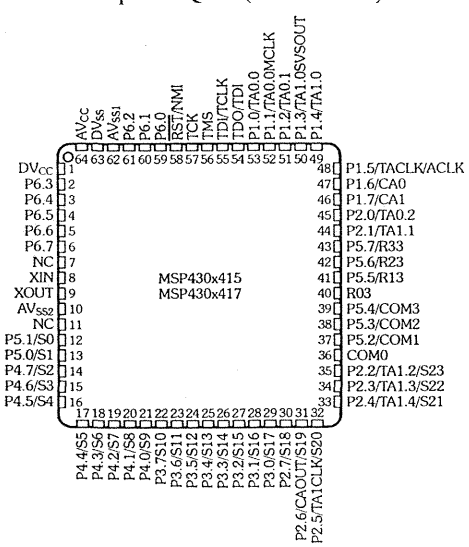
64 pinos QFN



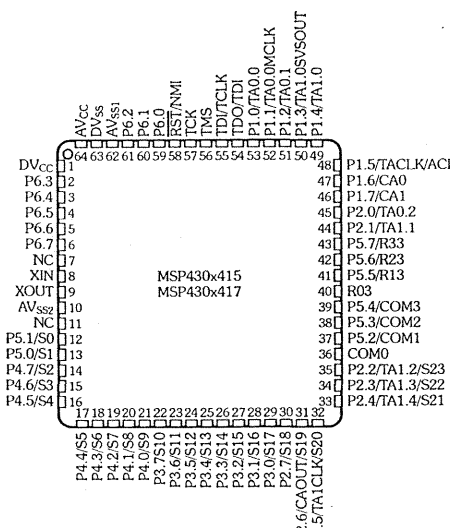
64 pinos LQFP/TQFP



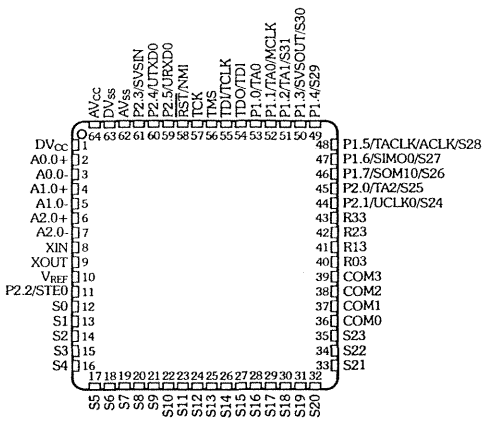
64 pinos QFN (família 41x)



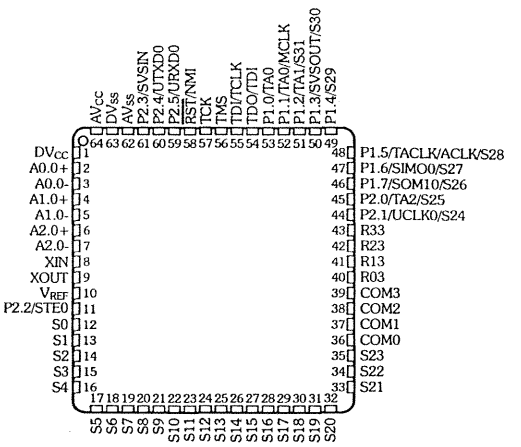
64 pinos LQFP/TQFP (família 41x)



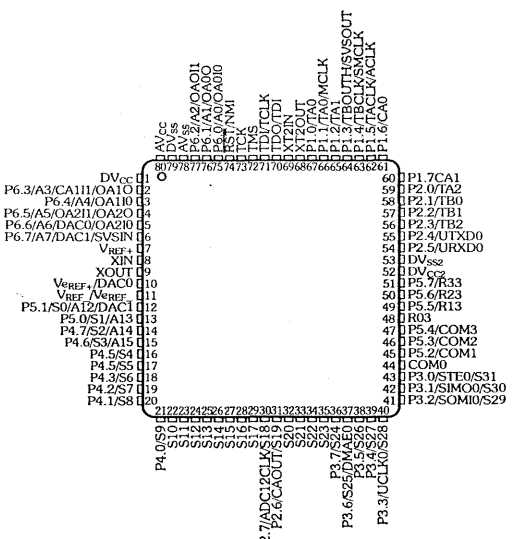
64 pinos QFN (família FE4xx)



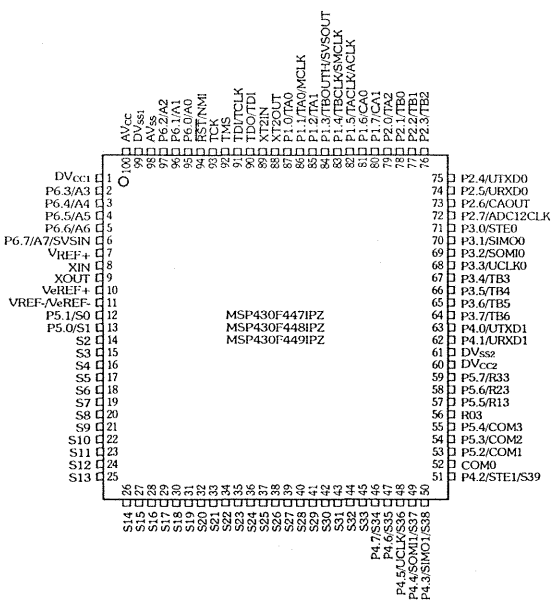
64 pinos LQFP/TQFP (família FE4xx)



80 pinos LQFP



100 pinos LQFP



Atenção: *Alguns dispositivos podem possuir uma pinagem ligeiramente diferente da apresentada. É importante consultar o* ***datasheet*** *do componente para se certificar da pinagem do modelo utilizado.*

3

O Ambiente *Embedded Workbench*

No decorrer do livro, vamos utilizar o ambiente *Embedded Workbench* da IAR *Systems* para o estudo dos microcontroladores MSP430, bem como para o desenvolvimento dos exemplos e aplicações, por isso, neste capítulo, vamos estudar em maiores detalhes o funcionamento e utilização desse poderoso ambiente de programação.

O ambiente *Embedded Workbench* é um IDE (***Integrated Development Environment*** - Ambiente Integrado de Desenvolvimento) composto de um editor de arquivos, montador *Assembly*, Compilador C e *Embedded* C++, ligador, simulador e emulador. Isso significa que o programador necessita de apenas uma ferramenta de *software* para todo o processo de desenvolvimento utilizando os microcontroladores MSP430.

Note que existem outras ferramentas disponíveis, tais como: MSPGCC e *Code Composer Essentials, Quadravox, Imagecraft e Crossworks (Rowley).*

Na figura 3-1 temos a janela inicial do *Embedded Workbench.*

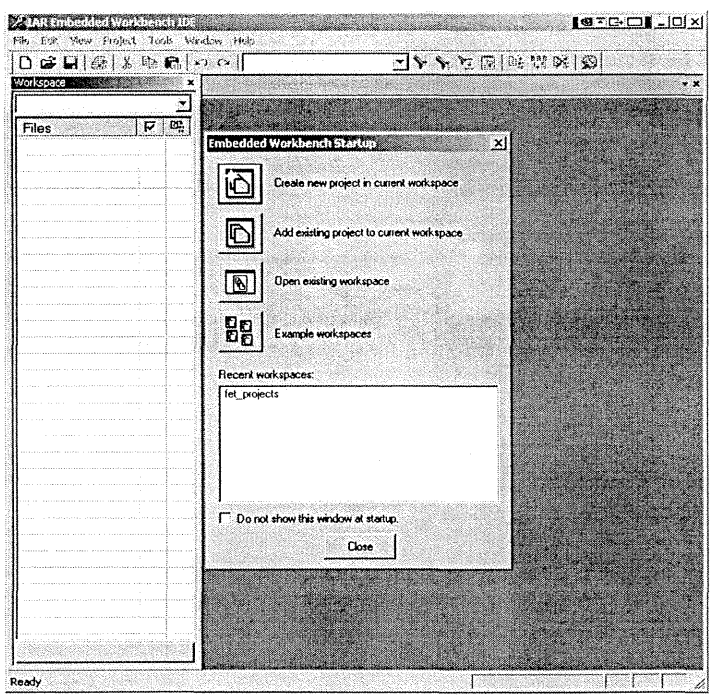


Figura 3-1

O ambiente é baseado no conceito de *workspaces* (espaços de trabalho) que são módulos que podem agregar um ou mais projetos. Um projeto pode conter um ou mais arquivos de código-fonte, utilizado(s) para gerar um arquivo binário, utilizável na simulação e programação do microcontrolador.

A janela central que aparece na figura 3-1 permite fazer a opção inicial entre quatro disponíveis:

 - criar um projeto dentro do *workspace* atual.

 - adicionar um projeto já existente ao *workspace* atual.

 - abrir um *workspace* já existente.

 - abrir um *workspace* de exemplos.

No decorrer deste capítulo, vamos verificar passo a passo como criar um projeto em linguagem *Assembly* e outro em linguagem C. Na tela da figura 3-1 o usuário deve selecionar a primeira opção (novo projeto).

3.1. Iniciando um Novo Projeto em *Assembly*

Ao selecionar a opção *Create New Project*, será apresentada a janela da figura 3-2 na qual o usuário informa o tipo de projeto que deseja criar.

Inicialmente vamos criar um projeto utilizando a linguagem *Assembly*, assim o usuário deve abrir a opção ASM e selecionar o item ASM, conforme a figura 3-3.

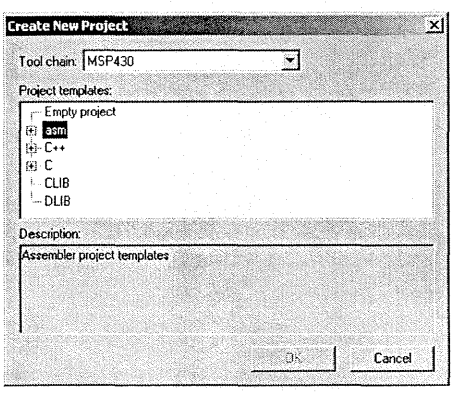


Figura 3-2

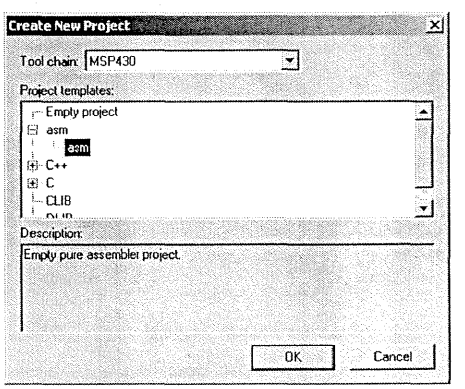


Figura 3-3

Selecionada a opção indicada na figura 3-3 e um clique no botão OK, será aberta uma janela solicitando o nome do arquivo para salvamento do projeto. Neste exemplo, vamos utilizar o nome **pisca_led_asm** (veja a figura 3-4).

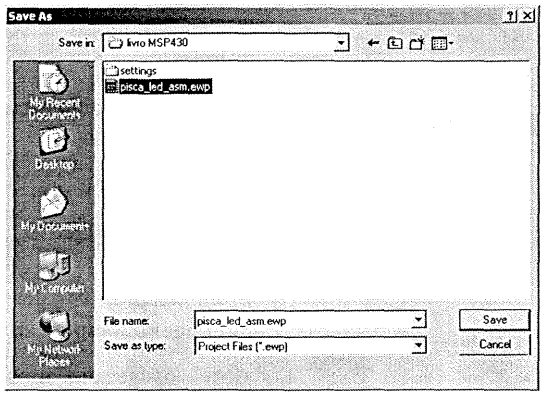


Figura 3-4

Criado e salvo o projeto, o *Embedded Workbench* deve estar com a aparência da figura 3-5.

A tela possui três partes básicas: a janela do *workspace*, que contém o projeto recém-criado, bem como os arquivos pertencentes a ele; a janela contendo o arquivo **asm.s43**, que contém o código-fonte *Assembly* do programa; e a janela *Debug Log*, que contém as informações sobre o processo de depuração do programa.

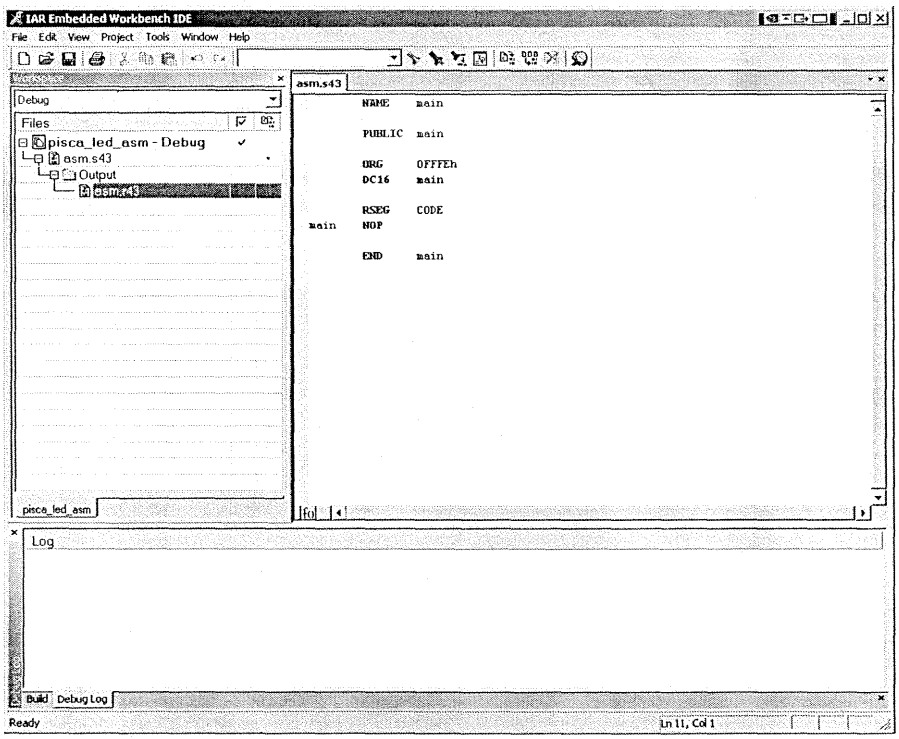


Figura 3-5

Vejamos em detalhes a janela do *workspace* atual:

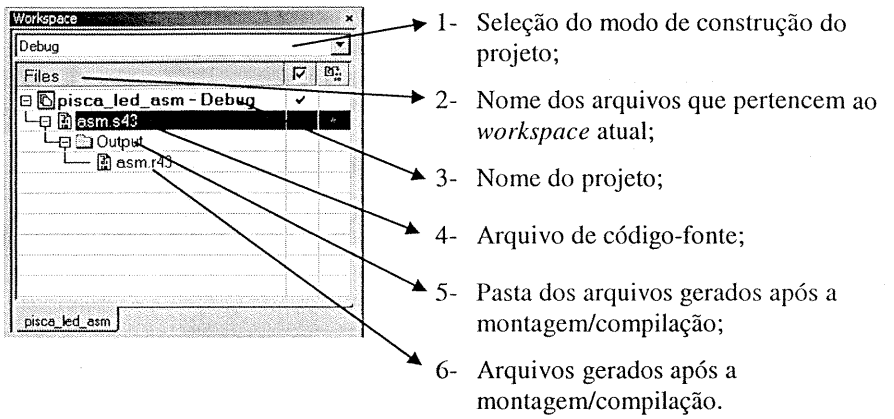


Figura 3-6

A opção 1 para seleção do modo de construção do projeto permite ao programador selecionar inicialmente uma entre duas opções possíveis: *DEBUG* ou *RELEASE*.

Cada uma das opções pode ser configurada para atender às necessidades específicas do programador (normalmente utiliza-se uma para a simulação e outra para a depuração do programa na aplicação). A configuração das opções do projeto é feita pela opção *PROJECT>OPTIONS* no menu principal, *Options...* no menu sensível ao contexto, ou ainda pressionando as teclas ALT F7.

Além do arquivo contendo o código-fonte do programa, os arquivos incluídos no projeto (que possuam vínculo de dependência com o projeto), tais como *headers* (extensão .H), também aparecerão listados na janela da figura 3-6.

Dentro da pasta *OUTPUT* (item 5 da figura 3-6), aparecem os arquivos gerados após a montagem/compilação do programa, incluindo os mapas de ligação e eventuais arquivos de listagem.

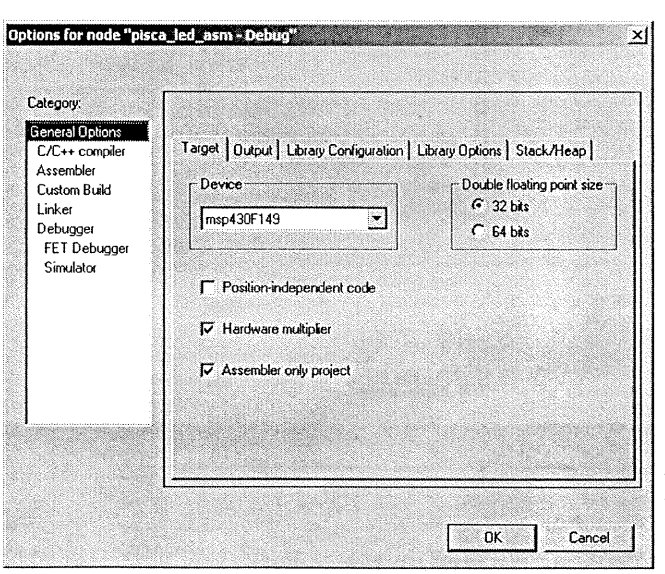


Figura 3-7

Com a janela de configuração do projeto aberta (tal qual a da figura 3-7), o programador pode selecionar e configurar diversas opções de montagem/compilação do projeto. Na categoria *General Options*, é possível configurar as opções gerais do projeto, na página *Target*, podemos selecionar o modelo do *chip* utilizado (na figura 3-7 está selecionado o MSP430F149), bem como a geração de código independente de posição (*Position-independent code*), habilitar a geração de código para o multiplicador por *hardware* (quando disponível no modelo de *chip* selecionado).

A opção *Assembler only project* pode ser utilizada quando se deseja programar somente em linguagem *Assembly*. Neste caso, fazendo a seleção dessa opção, o ambiente configura automaticamente as ferramentas para o uso da linguagem *Assembly* (desabilitando a utilização de bibliotecas, códigos de inicialização, etc.).

Ainda na categoria *General Options*, temos outras páginas: *Output*, que permite selecionar o tipo de arquivo gerado (executável ou um arquivo de biblioteca), *Library Configuration* e *Library Options*, para configuração da biblioteca utilizada na programação em C e *Stack/Heap*, que permite definir o tamanho da pilha de memória (*Stack*) e da área para armazenamento temporário (*Heap*). No caso de um projeto *Assembly*, somente a opção *Output* está disponível.

Na categoria *Assembler*, podemos configurar as opções do montador de programas (*Assembler*). A página *Language* configura algumas opções do montador, em *Output*, podemos selecionar o tipo de arquivo gerado. É importante que a opção *Generate debug information* esteja selecionada, de forma que seja gerada a informação de depuração que será utilizada pelo simulador/depurador.

Na página *List*, é possível selecionar e configurar a criação de arquivos de listagem que demonstram diversos aspectos do programa montado, incluindo os símbolos, macros, etc.

A página *Diagnostics* permite configurar a geração de mensagens de aviso pelo montador.

Na categoria *Linker*, temos as opções de configuração do ligador. Na página *Output*, configuramos o tipo de saída do ligador. É importante que o formato (*Format*) de saída seja *Debug information for C-SPY*, de forma que o simulador/depurador possa depurar corretamente o programa. Na página *List*, podemos selecionar a geração de um arquivo com o mapa de ligação, que demonstra a quantidade de memória e a posição ocupada por cada módulo, símbolo e variável do programa.

Finalmente, temos a categoria *Debugger*, que contém as opções da ferramenta de depuração. Na página *Setup*, podemos selecionar o *driver* de saída do depurador: Simulador (*Simulator*) ou o depurador FET (*FET Debugger*). O simulador pode ser utilizado para simular o funcionamento do programa, sem a necessidade de programar um *chip*. O depurador FET permite testar o programa, programando o *chip* e executando o programa diretamente nele.

Diretamente abaixo da categoria *Debugger*, temos as duas subcategorias referentes às duas ferramentas de depuração disponíveis: *FET Debugger* e *Simulator*, as quais permitem configurar opções específicas da ferramenta em questão.

Em *FET Debugger*, temos a opção de que seja feita a verificação do programa após a programação do *chip* (*Verify download*), utilização de pontos de parada virtuais (*Use virtual breakpoints*). Por padrão, a cada conexão de depuração, o *chip* é apagado e programado, no entanto, utilizando a opção *Supress download*, podemos evitar que o *chip* seja programado na conexão com o depurador e a opção *Ask when downloading* pode ser utilizada para questionar o programador sobre a programação ou não da memória do microcontrolador.

Também é possível especificar se a memória de programa e a memória de informação serão apagadas ou não antes da programação do *chip*.

Normalmente, o *driver* padrão do depurador é o simulador (*Simulator*) que permite que se teste o programa sem a necessidade de programar fisicamente um *chip*. O simulador (chamado C-SPY) encarrega-se de interpretar as instruções do programa e simular o funcionamento do *chip*.

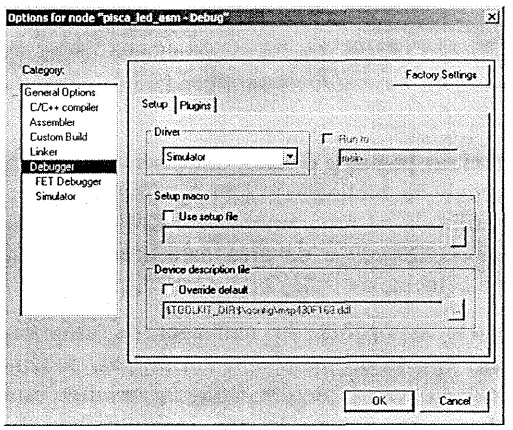


Figura 3-8

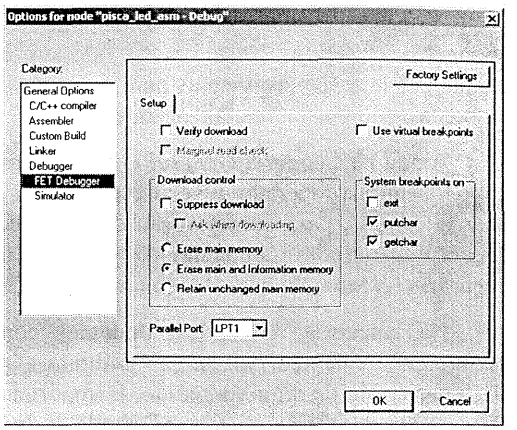


Figura 3-9

Com o projeto devidamente configurado, podemos passar à escrita do programa exemplo, que no nosso caso fará com que um LED com o ânodo conectado por meio de um resistor de 330 Ohms ao pino P1.0 e com o cátodo conectado ao terra, pisque em baixa freqüência.

O programa, propriamente dito, será escrito na janela **asm.S43**, que aparece na tela da figura 3-5, conforme a listagem seguinte. Repare que os termos em negrito são os que já aparecem previamente escritos no arquivo.

```
#include <msp430x14x.h>
        NAME    main
        PUBLIC  main
        ORG     0FFFEh
        DC16    main
        RSEG    CODE
main    MOV       #0x3FF,SP    ; inicializa o apontador da pilha
        MOV       #WDTPW+WDTHOLD,WDTCTL ; desliga o watchdog
        BIS.B     #1,P1DIR     ; configura o pino P1.0 como saída
loop    BIC.B     #1,P1OUT     ; limpa a saída P1.0
        CALL      #atraso      ; chamada da sub-rotina de atraso
        BIS.B     #1,P1OUT     ; seta a saída P1.0
        CALL      #atraso      ; chamada da sub-rotina de atraso
        BR        #loop        ; desvia para o loop
atraso  CLR       R10          ; apaga o conteúdo do registrador R10
atraso1 DEC       R10          ; decrementa R10
        TST       R10          ; testa se R10 é igual a zero
        JNZ       atraso1      ; se não for, desvia para atraso1
        REuT                   ; se for, retorna da sub-rotina
        END     main
```

Exemplo 3-1

3.1.1. Montando o Programa

Com o programa corretamente digitado, é chegada a hora de ordenar ao *Embedded Workbench* que monte o programa.

Para isso basta selecionar a opção *PROJECT>MAKE* no menu principal ou pressionar a tecla F7. Também é possível realizar a montagem clicando com o *mouse* no ícone na barra de ferramentas localizada logo abaixo do menu principal.

Com o processo de montagem concluído, o *Embedded Workbench* deve estar com a seguinte aparência:

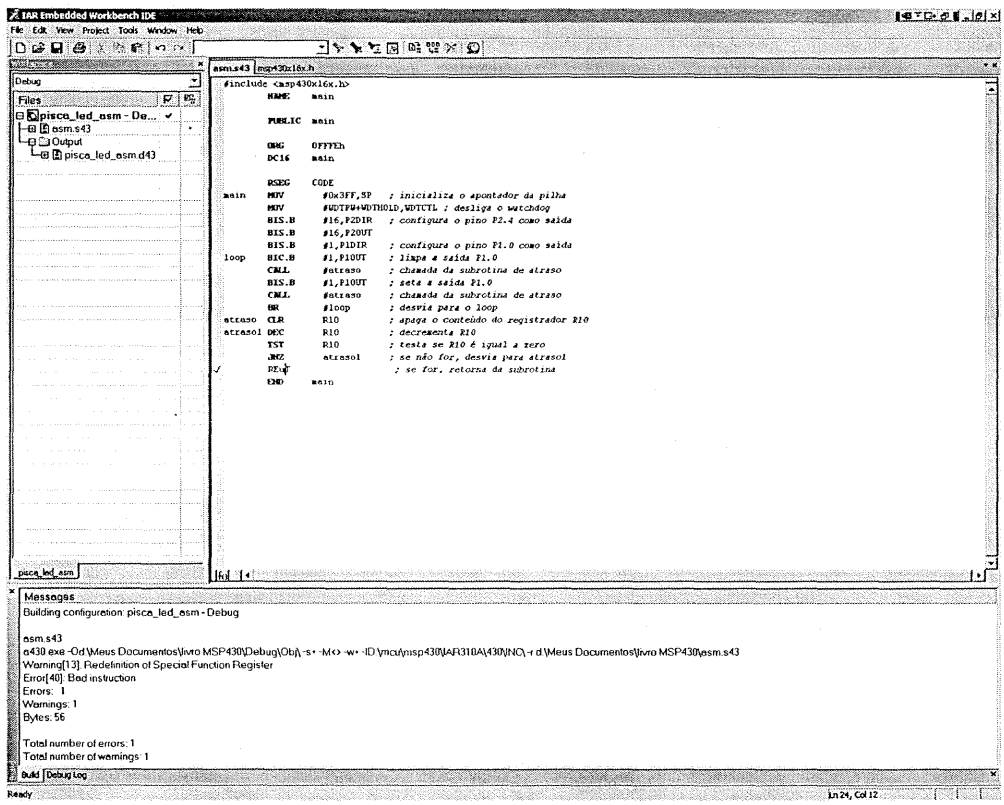


Figura 3-10

Repare que na parte inferior da tela aparecem diversas informações referentes ao processo de montagem/compilação do programa (na aba *Build*).

Ali também é apresentado o espaço total ocupado pelo programa.

As últimas linhas da janela de mensagens apontam a existência ou não de erros ou alertas gerados pelo *Assembler*.

Se o seu programa contiver algum erro, é sinal de que houve algum erro de digitação. No nosso caso houve um erro proposital: a instrução RET foi escrita como REuT. Neste caso, a janela inferior apresenta a mensagem: *Total number of errors: 1* e logo acima aparece *Error[40]: Bad instruction.*

Observe que um duplo-clique sobre a mensagem *Error[40]: Bad instruction* faz com que o cursor na janela de edição seja automaticamente movido para o ponto em que o erro foi localizado. A linha que contém o erro também encontra-se sinalizada com uma marca vermelha.

Corrigido o erro, basta clicar novamente no ícone ou pressionar a tecla F7 para que o processo de montagem ocorra novamente.

Com o processo concluído sem erros, a mensagem *Total number of errors: 0* deve ser apresentada. Repare que a existência de mensagens de alerta *(Total number of warnings: 1)* não significa um erro. Trata-se apenas de um aviso ao programador de que algo foi encontrado no programa que pode provocar falhas ou problemas.

Se nas opções do projeto o programador escolheu a geração de um arquivo de listagem do ligador (categoria *Linker*, página *List*, opção *Generate linker listing*), após a montagem do projeto, o arquivo .MAP estará listado na pasta *OUTPUT* dentro dos arquivos do projeto.

Os arquivos com extensão .XCL que também aparecem na pasta *OUTPUT* são os arquivos de controle do ligador e contêm as informações sobre a configuração do ligador para o projeto atual.

3.1.2. Simulando a Execução

Com o projeto devidamente montado, é hora de testar o programa no simulador. Para isso basta selecionar a opção *PROJECT > DEBUG* no menu principal, ou pressionar as teclas CTRL + D, ou ainda clicar no ícone , disponível na barra de ferramentas do *Embedded Workbench*.

O ambiente passa então a trabalhar no modo de depuração com a aparência da figura 3-11.

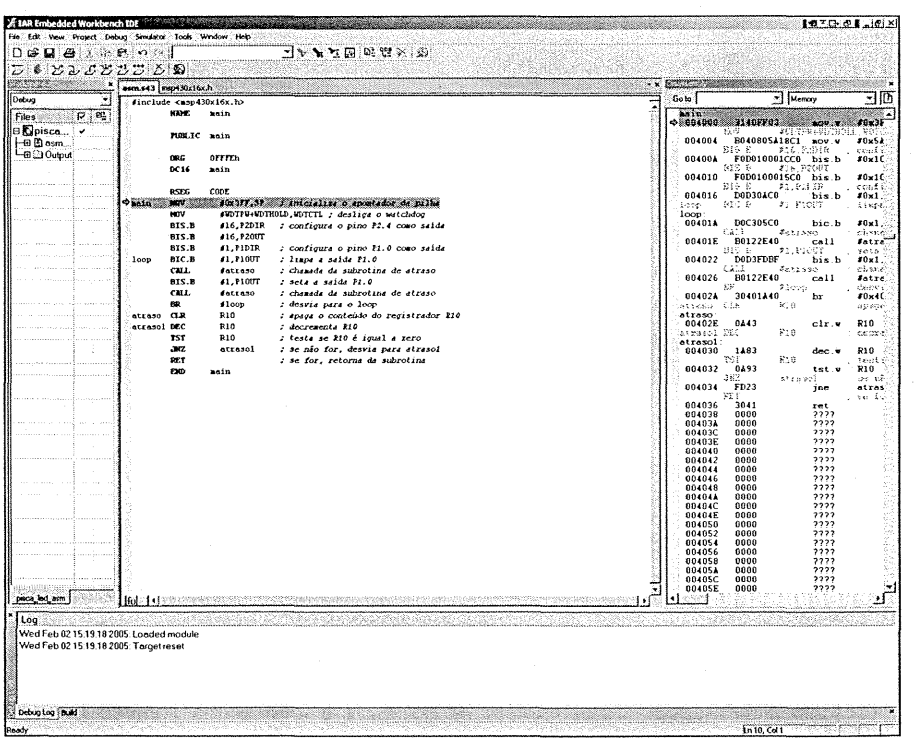


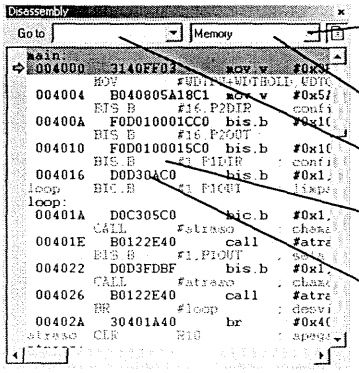
Figura 3-11

No modo de depuração encontram-se disponíveis as diversas opções para execução e depuração do programa. Como o projeto atual está configurado para trabalhar com o simulador, a depuração é feita com o simulador.

Antes de seguirmos com os procedimentos da simulação propriamente dita, vejamos alguns detalhes sobre a interface gráfica do *Embedded Workbench* quando opera no modo de depuração.

A primeira coisa que vemos ao entrar no modo de depuração é a janela adicional *Disassembly* que é aberta na lateral direita do programa (figura 3-12).

A janela exibe o conteúdo da memória do microcontrolador tanto no formato hexadecimal como intercalado com o código-fonte original (*Assembly* ou C).



1- Seleção do modo de exibição: apenas o código objeto desmontado, ou o código objeto intercalado com o código-fonte original;

2- Seleção da área da memória a ser exibida;

3- Seleção do endereço ou rótulo a ser apresentado;

4- Linha do código-fonte original;

5- Código objeto na memória.

Figura 3-12

O ambiente permite configurar a disposição das janelas e estão disponíveis janelas com capacidade de apresentar os mais diversos conteúdos. A modificação da posição de uma janela é feita com um clique na sua barra de título e em seguida ela é arrastada para a posição desejada. Para adicionar uma nova janela, basta selecionar a opção *VIEW* no menu principal e escolher uma das janelas de visualização disponíveis. As principais estão listadas em seguida:

- *Disassembly* - já discutida anteriormente.
- *Memory* - para visualizar a memória sem a desmontagem de código, mas com diversas opções adequadas à visualização de dados.
- *Register* - utilizada para visualização do conteúdo dos diversos registradores tanto da CPU quanto dos SFRs localizados na RAM.
- *Watch* - visualização de expressões ou variáveis definidas pelo usuário.
- *Locals* - permite visualizar as variáveis locais (dentro de uma sub-rotina ou função).
- *Live Watch* - visualização de expressões ou variáveis definidas pelo usuário, mas em tempo real, mesmo durante a execução de um programa. A taxa de atualização (que por padrão é de 1.000 milissegundos) pode ser alterada pelo usuário (selecionando a opção *TOOLS > OPTIONS* e em seguida a aba *Debugger*, no menu principal).
- *Call Stack* - permite a visualização do conteúdo da pilha de chamada de funções C, apresentando a lista de funções encadeadas, com a última função chamada no topo.
- *Terminal I/O* - utilizada para depuração de programas. Essa janela recebe o conteúdo dos dispositivos *stdin*, *stdout* e *stderr*, e pode ser utilizada para receber e enviar

informação para a aplicação. Para a sua utilização é necessário ativar a opção do ligador: *With I/O emulation modules*.

- *Code Coverage* - a janela de cobertura de código permite analisar quais partes do programa são mais executadas e quais são menos executadas, permitindo ao programador otimizar as partes críticas da aplicação.
- *Profiling* - permite a medição do tempo gasto na execução das funções do programa.
- *Trace* - para visualização de informações do rastreamento do programa.
- *Stack* - permite visualizar o conteúdo da pilha de memória.

A janela *Register* merece uma atenção especial, já que nela estão concentrados dados essenciais para a depuração de um programa.

Observando a janela da figura 3-13, podemos verificar que estão presentes todos os registradores existentes na CPU. Os registradores mapeados *bit* a *bit* podem ser expandidos para visualização detalhada do conteúdo de cada um dos seus *bits* ou conjunto de *bits*. Na figura em questão, podemos ver que o registrador de *status* (SR) está expandido, permitindo a observação (e alteração) do conteúdo de cada um dos *bits* que o compõem.

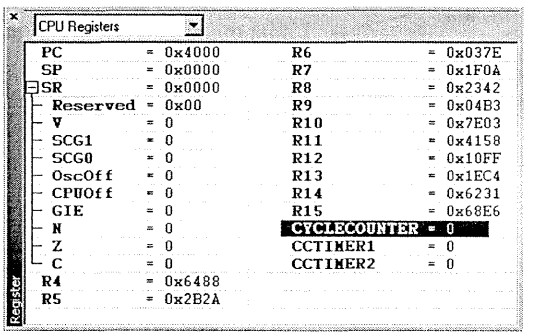


Figura 3-13

Na figura 3-13, podemos ver também o contador de ciclos (*CYCLECOUNTER*) utilizado para a contagem do número de ciclos de *clock* gastos na execução do programa. Essa facilidade pode ser útil quando desejamos medir a quantidade de tempo gasto na execução de uma determinada tarefa ou em uma determinada porção de código.

Além dos registradores da CPU, também é possível visualizar outros registradores, que se encontram agrupados de acordo com o periférico a que pertencem. Assim, temos um grupo de registradores do módulo oscilador, outro dos registradores de funções especiais (SFRs), outro grupo para as portas 1 e 2 de entrada/saída e assim por diante. A disponibilidade de um ou mais grupos de registradores está relacionada à disponibilidade de cada periférico no *chip* selecionado no projeto.

Também é possível ter múltiplas instâncias dessa janela abertas simultaneamente, o que permite que se visualizem diversos grupos de registradores ao mesmo tempo.

Outro detalhe importante e que não é válido somente para a janela *Register* é que a cada alteração do conteúdo de um registrador o seu valor aparece grafado em vermelho.

Além disso, podemos alterar o conteúdo de um registrador simplesmente clicando sobre o seu conteúdo prévio, digitando o novo valor, seguido da tecla *ENTER*.

Um aspecto interessante é que podemos criar grupos para visualização de registradores. Dentro das opções da IDE (opção *TOOLS > OPTIONS* no menu principal), basta selecionar a aba *Register Filter* e em seguida ativar a opção *Use register filter*. Feito isso, o usuário pode escolher um arquivo de filtragem já existente ou simplesmente digitar o nome do novo arquivo de filtragem. Considerando a criação de um arquivo, basta criar o grupo de registradores (botão *New Group...*) e em seguida selecionar os arquivos que se pretenda incluir no novo grupo.

A figura 3-14 demonstra a criação de um grupo chamado "Teste", contendo os registradores SR, IFG1, IFG2, P1OUT e P1DIR.

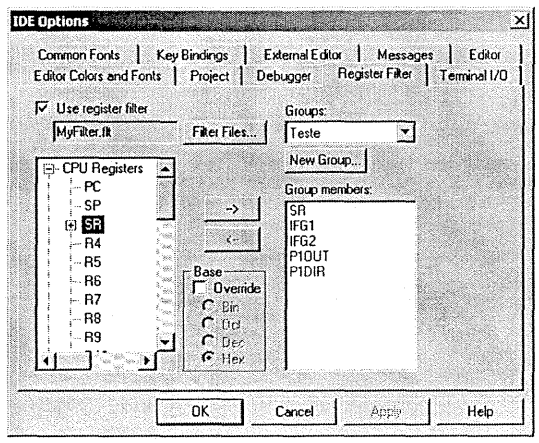


Figura 3-14

Depois de criado o novo grupo e desde que a opção *Use register filter* esteja ativada, o usuário deve ser capaz de visualizar o novo grupo de registradores na janela *Register*, como se fosse um dos outros grupos preexistentes. A figura 3-15 apresenta a visualização do novo grupo "Teste" criado anteriormente.

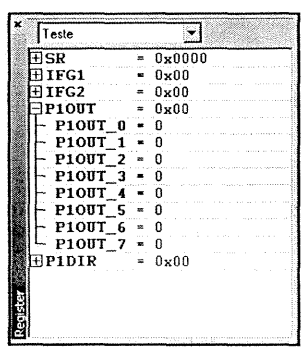


Figura 3-15

Na figura 3-16, podemos ver um exemplo de organização das janelas do *Embedded Workbench* quando opera no modo de depuração.

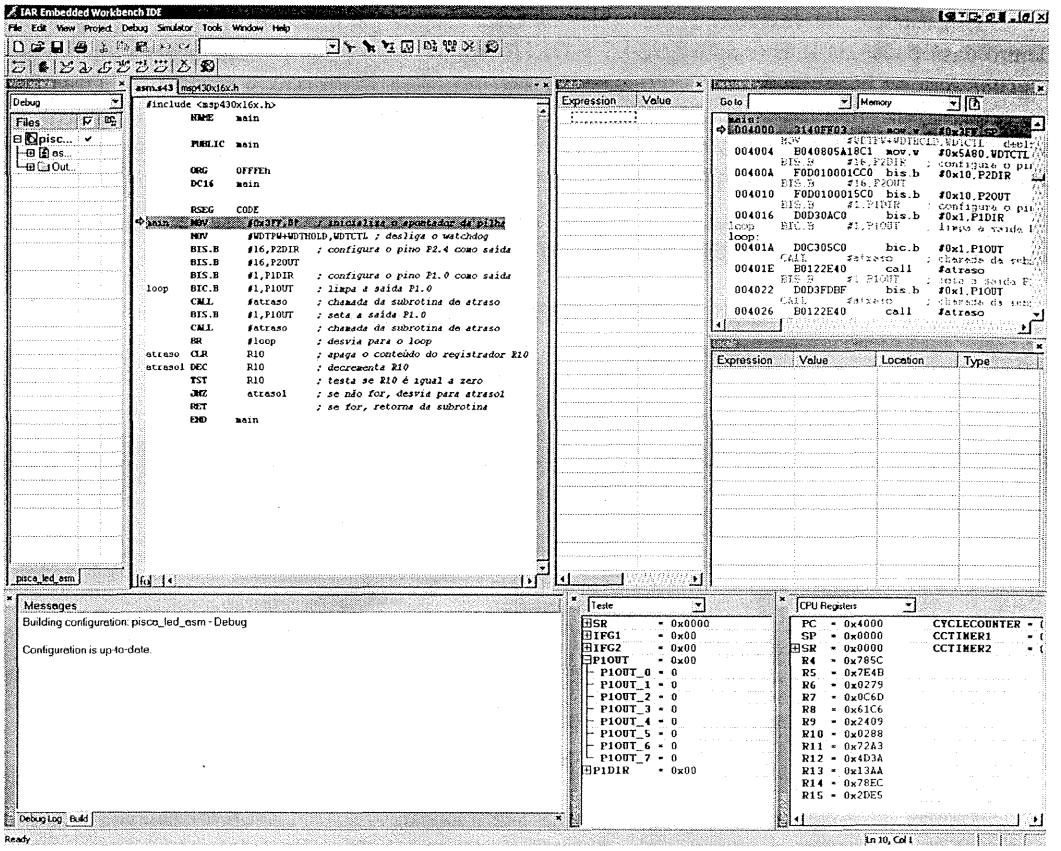


Figura 3-16

Além das janelas de visualização, o modo de depuração acrescenta uma nova barra de ferramentas ao IDE: a barra de depuração, com a qual o programador pode interagir com o programa em simulação/depuração, permitindo a sua execução contínua, passo a passo, parada, etc.

Esses controles também estão disponíveis no menu DEBUG e por meio das teclas de atalho apresentadas no mesmo menu e descritas mais adiante.

Antes de iniciarmos a simulação do programa, vamos configurar uma janela de visualização *Register* de forma a observarmos o comportamento dos registradores P1DIR e P1OUT, utilizados no programa exemplo para o controle do LED.

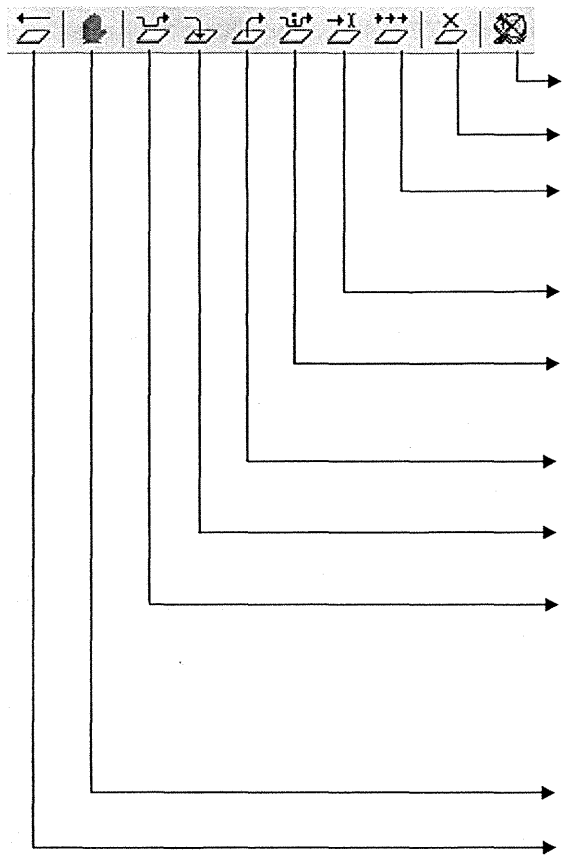
Para inserirmos uma nova janela de visualização, basta clicar na opção *VIEW > REGISTER* no menu principal.

Em seguida, devemos configurar a janela de forma que ela apresente o conteúdo dos registradores referentes à porta 1. Uma alternativa seria utilizar o grupo "Teste" criado há pouco (veja a figura 3-15).

Quando tudo estiver configurado, é chegada a hora de simularmos a execução do programa.

Podemos observar, na tela da figura 3-16, a existência de uma seta (verde), indicando uma linha do programa que também se encontra escrita sobre um fundo verde. É a indicação do

depurador de que esta é a próxima linha de programa que será executada. A mesma informação também pode ser vista na janela *Disassembly*.



1- Saída do modo de depuração;
2- Cria/apaga um *breakpoint*;
3- Executa o programa até um ponto de parada (*breakpoint*) ou o término do programa;
4- Executa o programa até o ponto em que se encontra o cursor;
5- Pula para o próximo comando (o atual é executado sem visualização no depurador);
6- Executa o programa até o término da função atual;
7- Executa a próxima instrução ou comando (passo a passo);
8- Executa a próxima instrução ou comando (se for uma chamada de sub-rotina ou função, ela é inteiramente executada e o controle retorna para a instrução seguinte à chamada);
9- Pára a execução do programa;
10- *Reset* do processador.

Figura 3-17

Nesse momento, pressionando a tecla F11, podemos proceder à execução da instrução **MOV #0x3FF,SP** que fará com que o valor 0x3FF seja copiado para o registrador SP.

Se observarmos o conteúdo desse registrador na janela *Register*, podemos verificar que se encontra grafado em vermelho, no entanto o valor apresentado é 0x3FE e não 0x3FF, como comandou a instrução.

A razão para isso é muito simples e já foi vista no capítulo anterior: o registrador SP (R1) mantém o seu *bit* 0 (LSB) sempre com o valor "0", fazendo com que ele somente aponte para endereços pares de memória. Se apagarmos o *bit* menos significativo do valor 0x3FF, chegamos a 0x3FE que é o valor efetivamente armazenado no SP.

Além do registrador SP, podemos verificar que também o registrador PC teve o seu conteúdo alterado. Isso ocorre porque, após a execução da primeira instrução, o conteúdo do PC é incrementado de forma a apontar para a próxima instrução a ser executada.

Observando a tela do *Embedded Workbench*, podemos verificar que a próxima instrução a ser executada (a que é apontada pelo PC) é a localizada no endereço 0x4004 (podemos ver isso na janela *Disassembly*): **MOV #WDTPW+WDTHOLD,** WDTCTL. O fato de a instrução aparecer

grafada como MOV.W na janela *Disassembly* não deve causar nenhum espanto. Isso ocorre porque MOV e MOV.W são sinônimos, como veremos no próximo capítulo, no estudo das instruções do MSP430.

Pressionando a tecla F11 novamente (comando *STEP INTO*, equivalente a clicar no ícone 7 da figura 3-17), provocaremos a execução do comando, que fará com que o conteúdo do registrador WDTCTL seja alterado.

As instruções seguintes configuram o pino P1.0 como saída e o seu estado como inativo (nível "0").

Para avançar a execução do programa até a chamada da sub-rotina "atraso", o leitor pode se utilizar do seguinte artifício: coloque o ponteiro do mouse sobre a instrução **CALL #atraso**, dê um clique com o botão direito e no menu que surgir, clique na opção *RUN TO CURSOR*.

O programa será executado até alcançar aquela instrução e pára nela.

Neste ponto, podemos seguir por dois caminhos distintos: executar a sub-rotina "atraso" sem o acompanhamento da sua execução (utilizando a função *STEP OUT* ou as teclas SHIFT + F11), ou seguir o programa normalmente e acompanhar a execução da sub-rotina (utilizando a função *STEP INTO* ou a tecla F11).

Vamos optar por acompanhar a execução dessa sub-rotina e pressionar a tecla F11 de forma que isso aconteça.

Observe que o fluxo do programa é desviado para o rótulo "atraso", que equivale ao endereço 0x402E da memória. Outro detalhe importante é que o SP foi decrementado de 2, já que o endereço de retorno da sub-rotina foi salvo na pilha.

Se o leitor tiver curiosidade sobre o funcionamento desse processo, pode abrir uma janela de visualização da pilha (opção *VIEW>STACK* no menu principal), na qual haverá uma seta verde indicando o topo da pilha e nesse local está armazenado o valor 0x4022, que nada mais é do que o endereço da instrução **BIS.B #1,P1OUT** na memória (veja a figura 3-18).

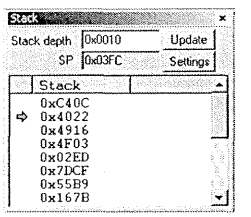


Figura 3-18

Prosseguindo na execução passo a passo do programa (tecla F11), o leitor pode observar que inicialmente o conteúdo do registrador R10 será apagado e em seguida decrementado e testado contra o valor zero. Se diferente de zero, o programa é desviado para o rótulo "atraso1".

Após algum tempo experimentando a execução passo a passo, o leitor deve perceber que gastará muito tempo até que o conteúdo do registrador R10 atinja o valor zero.

Para abreviar esse tempo, podemos utilizar dois artifícios: *RUN TO CURSOR*, ou criar um *breakpoint* no local desejado para a parada do programa.

Vamos utilizar a segunda opção e criar um *breakpoint* na instrução de retorno (**RET**). Basta clicar na instrução com o botão direito do *mouse* e em seguida selecionar a opção *TOGGLE BREAKPOINT*. A janela contendo o código-fonte deve apresentar uma linha com fundo vermelho, como a da figura 3-19.

O ponto de parada (*breakpoint*) nada mais é do que um ponto do programa que, ao ser atingido, provoca a parada da sua execução.

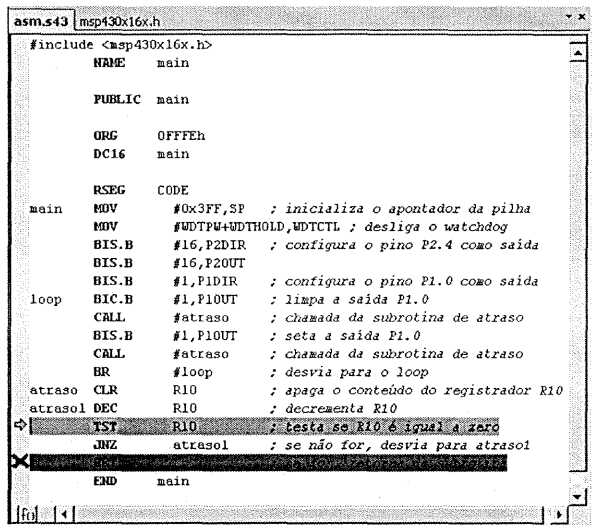


Figura 3-19

Nesse momento podemos comandar a execução contínua do programa (comando *GO*, equivalente ao ícone 3 da figura 3-17 ou à pressão da tecla F5). O programa será rapidamente executado até que seja atingido o *breakpoint*.

Ao atingir o *breakpoint*, a execução é paralisada e podemos observar o conteúdo dos registradores: R10 contém o valor 0, como esperado, e o *bit* Z do registrador SR está setado, indicando que o resultado do teste realizado pela instrução TST foi verdadeiro.

Pressionando novamente a tecla F11, faremos com que o fluxo do programa seja desviado para o endereço de retorno da sub-rotina (a instrução **BIS.B #1,P1OUT**).

A execução dessa instrução provoca a ativação do pino P1.0 (fazendo com que o valor de P1OUT_0 na janela de visualização *Register* seja igual a "1").

Prosseguindo com a execução do programa, o leitor pode observar que o estado de P1OUT_0 permanece alternando entre "0" e "1".

Uma última experiência que vamos fazer antes de encerrar esta rápida discussão sobre a simulação do programa é a utilização do *Live Watch*. Este é um recurso muito interessante e que permite acompanhar o estado de variáveis durante a execução do programa, sem a necessidade de pará-lo.

Para criar uma janela *Live Watch*, selecione a opção *VIEW* > *LIVE WATCH* no menu principal. Na coluna *Expression* da janela, digite: #P1OUT e então pressione ENTER.

Para podermos visualizar corretamente a freqüência da mudança de estado desse *bit*, devemos configurar um novo valor de atualização para o *Live Watch*. Isso pode ser feito clicando com o botão direito do *mouse* no interior da janela e selecionando a opção *OPTIONS* no menu.

Um valor de 100 milissegundos para o campo *Update interval* é mais que suficiente.

Configurado o *Live Watch*, basta iniciar a execução do programa para acompanhar a alternância de estado do sinal P1OUT (será necessário clicar sobre a instrução **RET** e selecionar a opção *TOGGLE BREAKPOINT* para desativar o *breakpoint* naquele local).

Em tempo: para parar a execução do programa, basta clicar no ícone 9 da figura 3-17 (comando *Break*).

3.2. Iniciando um Novo Projeto em C

O procedimento para iniciar um novo projeto utilizando a linguagem C é bastante parecido com o já visto para a linguagem *Assembly*. A figura 3-20 demonstra a seleção da linguagem C para o novo projeto.

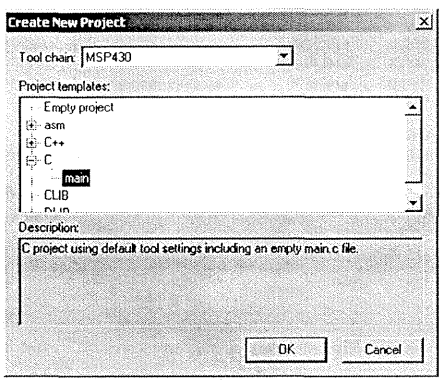


Figura 3-20

Após ser dado um nome para o arquivo do projeto, surge uma tela semelhante à da figura 3-5, porém o arquivo para edição possui o nome **main.c**.

Em seguida o leitor pode proceder à digitação do programa exemplo, conforme a listagem seguinte (os comandos grafados em negrito já se encontram previamente escritos no arquivo main.c).

```
#include <msp430x14x.h>
#include <stdio.h>
void atraso(void)
{
   unsigned int temp;          // declara a variável temp
   temp=0;                     // inicializa temp em 0
   do temp--; while (temp);    // laço de contagem regressiva até 0
}
int main( void )
{
    WDTCTL = WDTPW + WDTHOLD;  // desativa o watchdog
    P1DIR = 0x01;              // configura o pino P1.0 como saída
    while (1)                  // loop infinito
    {
       P1OUT = 0;              // apaga os bits da porta P1
       //printf("desligado\r\n");      // escreve no terminal de I/O
       atraso();               // chama a função de atraso
       P1OUT = 1;              // seta o pino P1.0
       //printf("ligado\r\n");  // escreve no terminal de I/O
       atraso();               // chama a função de atraso
    }
}
```

Exemplo 3-2

O projeto em linguagem C disponibiliza ainda uma série de novas opções em termos de configuração, além daquelas já vistas para um projeto em *Assembly*. Vejamos algumas das principais opções disponíveis para o projeto em C:

- *Library Configuration* - permite escolher o tipo de biblioteca padrão a ser ligada ao projeto. A biblioteca padrão contém as principais funções definidas no C ANSI. Maiores detalhes sobre as bibliotecas padrão serão vistos quando estudarmos o compilador C IAR no tópico 6.2.
- *Library Options* - utilizada para especificar o grau de complexidade de implementação das funções *printf* e *scanf*, influenciando diretamente no tamanho do arquivo gerado.
- *Stack/Heap* - essa página permite configurar o tamanho da pilha de memória (*Heap*) e da área de armazenamento dinâmico (*Heap*).

Na categoria C/C++ *Compiler*, temos como principais opções as seguintes:

- *Language* - nessa página podemos especificar alguns detalhes sobre a forma como a linguagem C é interpretada, incluindo o suporte ao *Embedded C++* e o tipo padrão para o **char**.
- *Code* - utilizada para configurar as opções de otimização de código pelo compilador.
- *Output* - para configurar os tipos de arquivo de saída (aqui encontramos a opção *Generate debug info*, necessária para a depuração de programas).
- *List* - permite configurar os tipos de arquivo de listagem gerados pelo compilador, incluindo a geração de arquivos *Assembly*.

As demais opções são as mesmas já apresentadas para um projeto *Assembly* e não serão novamente discutidas.

3.2.1. Compilando um Projeto em C

A compilação do programa pode ser iniciada selecionando a opção *PROJECT>COMPILE* no menu principal ou pressionando as teclas CTRL + F7. Também é possível realizar a compilação clicando com o mouse no ícone  na barra de ferramentas localizada logo abaixo do menu principal, ou ainda, por meio da construção do projeto (como no projeto em *Assembly*). Basta selecionar a opção *PROJECT>MAKE* no menu principal ou pressionar a tecla F7.

As mensagens geradas pela compilação do programa são idênticas às geradas pela montagem de um programa *Assembly* e já foram discutidas anteriormente.

3.2.2. Simulando um Projeto em C

De maneira geral, a simulação de um programa em C não difere muito da simulação de um programa em *Assembly*. Neste tópico veremos apenas alguns detalhes importantes e únicos acerca de simulação/depuração de um programa em C.

A primeira característica importante a ser destacada é que a execução do programa em C pode ser feita com base nos comandos de alto nível ou com base nas instruções *Assembly*.

A opção entre um e outro modo é feita de acordo com a janela utilizada para acompanhar a execução: se clicarmos na janela que contém o código-fonte, a execução será feita com base nos comandos C; se clicarmos na janela *Disassembly*, a execução será feita seguindo os comandos *Assembly*.

Uma outra característica interessante é que podemos visualizar na janela *Disassembly* tanto apenas o código *Assembly* quanto o código-fonte em C intercalado com o código *Assembly* equivalente.

Também a visualização do conteúdo das variáveis é bastante facilitada no *Embedded Workbench*. Além das janelas *WATCH, LOCALS, AUTO* e *LIVE WATCH*, já comentadas anteriormente, também podemos observar o conteúdo de uma variável simplesmente posicionando o cursor do mouse sobre o nome dela no código-fonte.

Finalmente, podemos utilizar comandos como *STEP OUT* para forçar que a execução avance até o final da função em que o programa atualmente se encontra.

3.2.3. Terminal de I/O

Podemos utilizar o terminal de I/O disponível no menu *View* do depurador para visualizar a saída de funções como **printf**, **putchar**, etc.

No exemplo 3-2, dois comandos **printf** foram propositalmente deixados comentados (as linhas com // na sua frente). Caso desejado, o leitor poderá suprimir as barras de forma que os comandos **printf** passem a poder ser executados.

Além disso, é necessário ativar a opção do ligador que conecta as funções C ao terminal de I/O (opção *With I/O emulation modules* na janela de opções do projeto).

Na figura 3-21, podemos observar a configuração do ligador para o projeto utilizando o terminal de I/O, enquanto que na figura 3-22, temos o próprio terminal de I/O apresentando a impressão gerada pela execução de uma iteração do comando **while** do exemplo 3-2.

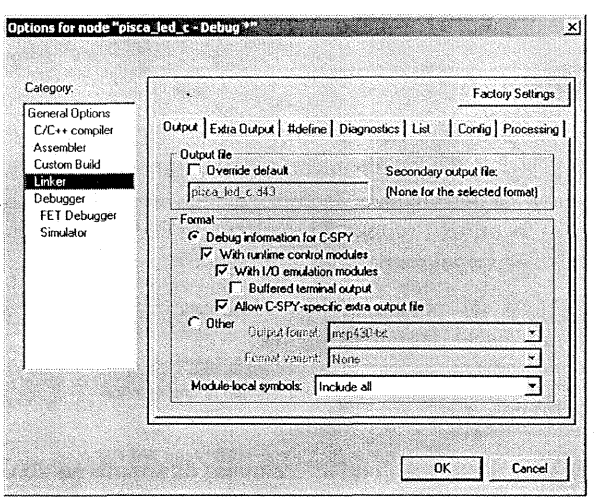


Figura 3-21

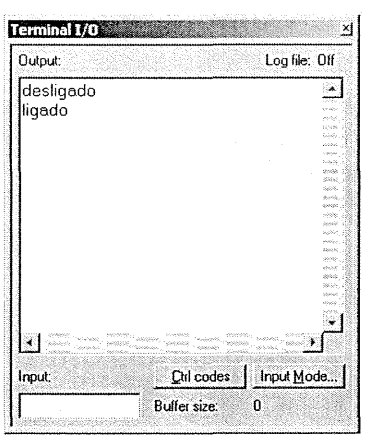


Figura 3-22

Um aspecto interessante da utilização desta abordagem é que também podemos utilizar o campo *input* da janela do terminal de I/O para fornecer dados para funções de entrada como **scanf**, por exemplo.

4

Instruções *Assembly*

Como já foi visto anteriormente, os MSP430 possuem um conjunto de 51 instruções, compostas de 27 instruções físicas e mais 24 instruções emuladas.

As instruções podem acessar a memória tanto no modo de 8 *bits* quanto no modo de 16 *bits*. Isso permite que se otimize o consumo de memória (diferenciando o acesso a variáveis de 8 ou 16 *bits*), além de propiciar um aumento de performance.

Este capítulo explica o conjunto completo de instruções, o funcionamento e utilidade das instruções emuladas, além dos diferentes modos de endereçamento disponíveis na arquitetura.

4.1. Construção dos *Op-codes*

Os *op-codes*, ou seja, os códigos binários que formam uma instrução, são classificados pelo fabricante em três categorias diferentes: instruções com um operando, instruções com dois operandos e instruções de desvio.

O tamanho de uma instrução varia entre uma e três *words*, com o *byte* menos significativo alinhado à esquerda.

O formato típico de uma instrução de três *words* é:

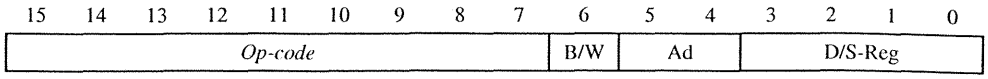
Endereço	Conteúdo
X	LSB do *op-code*
X + 1	MSB do *op-code*
X + 2	LSB do operando 1
X + 3	MSB do operando 1
X + 4	LSB do operando 2
X + 5	MSB do operando 2

Tabela 4-1

4.1.1. Instruções com Um Operando

As instruções com apenas um operando possuem um tamanho de uma a duas *words*. A primeira *word* especifica o *op-code* e eventualmente o registrador fonte/destino (no caso de instruções com tamanho de uma *word*). Quando a fonte/destino é uma posição da memória, a segunda *word* é utilizado para especificar a posição da memória.

O formato binário desse tipo de instrução é o seguinte:



Sendo:

Op-code - é o código binário que especifica a instrução desejada.

B/W - é o *bit* especificador do formato de acesso da instrução:

0 - acesso de 8 *bits*;

1 - acesso de 16 *bits*.

Ad - são os *bits* especificadores do formato de endereçamento da fonte ou destino da operação:

00 - o destino/fonte é o registrador especificado por D/S-Reg;

01 - o destino/fonte é o endereço resultante da soma do registrador indicado por D/S-Reg mais o operando especificado pela *word* seguinte;

10 - o destino/fonte é o endereço apontado pelo registrador especificado em D/S-Reg;

11 - o destino/fonte é o endereço apontado pelo registrador especificado em D/S-Reg.

Maiores detalhes sobre o funcionamento dos modos de endereçamento serão vistos no tópico 4.2.

D/S-Reg - são os *bits* que especificam qual dos 16 registradores da CPU será utilizado como fonte e/ou destino da operação.

Pertencem a esta categoria as seguintes instruções: CALL, PUSH, RETI, RRA, RRC, SWPB e SXT.

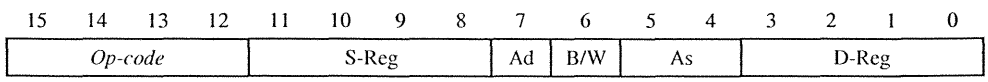
4.1.2. Instruções com Dois Operandos

A segunda categoria de instruções dos MSP430 possui dois operandos distintos chamados genericamente de fonte e destino.

Esse tipo de instrução possui o tamanho de uma a três *words*, dependendo dos operandos envolvidos:

- 1 *word* - caso ambos os operandos sejam registradores da CPU;
- 2 *words* - caso um dos operandos seja um registrador da CPU (o outro operando será um endereço ou constante imediata);
- 3 *words* - caso ambos os operandos sejam endereços de memória, ou um deles uma constante imediata.

O formato binário desse tipo de instrução é:



Sendo:

Op-code - é o código binário que especifica a instrução desejada.

S-Reg - são os *bits* que especificam qual dos 16 registradores da CPU será utilizado como fonte da operação.

Ad - é o *bit* especificador do formato de endereçamento do destino da operação:

0 - o destino é um registrador da CPU;

1 - o destino é um endereço da memória especificado pelo terceiro *word*.

Maiores detalhes sobre o funcionamento dos modos de endereçamento serão vistos no tópico 4.2.

B/W - é o *bit* especificador do formato de acesso da instrução:

0 - acesso de 8 *bits*;

1 - acesso de 16 *bits*.

As - são os *bits* especificadores do formato de endereçamento da fonte da operação:

00 - a fonte é o registrador especificado por S-Reg;

01 - a fonte é o endereço resultante da soma do registrador indicado por S-Reg com o operando especificado pelo *word* seguinte ao *op-code*;

10 - a fonte é o endereço apontado pelo registrador especificado em S-Reg;

11 - a fonte é o endereço apontado pelo registrador especificado em S-Reg.

Maiores detalhes sobre o funcionamento dos modos de endereçamento serão vistos no tópico 4.2.

D-Reg - são os *bits* que especificam qual dos 16 registradores da CPU será utilizado como destino da operação.

Pertencem a esta categoria as seguintes instruções: ADD, ADDC, AND, BIC, BIS, BIT, CMP, DADD, MOV, SUB, SUBC e XOR.

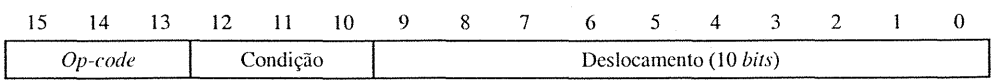
4.1.3. Instruções de Desvio

A terceira e última categoria de instrução é formada pelas instruções de desvio. Essas instruções são utilizadas para provocar um desvio relativo do programa.

O desvio ou salto relativo é realizado somando um valor de deslocamento sinalizado ao PC, de forma a originar o endereço de destino no programa.

Nos MSP430, o valor de deslocamento pode variar entre –511 e +512. Um detalhe importante é que o deslocamento é feito de 16 em 16 *bits* (dois em dois *bytes*), de forma que um deslocamento de +20 implica na soma do PC com o valor 40.

O formato binário desse tipo de instrução é ilustrado a seguir:



Sendo:

Op-code - é o código binário que especifica a instrução desejada, neste caso, o código é sempre 001.

Condição - o código de três *bits* que especifica a condição em que se realizará o desvio:

000 - desvia se não igual (diferente);

001 - desvia se igual;

010 - desvia se o *carry* igual a zero (C=0);

011 - desvia se o *carry* igual a um (C=1);

100 - desvia se negativo (N=1);

101 - desvia se maior ou igual (N = V ou (N XOR V = 0));

110 - desvia se menor (N ≠ V ou (N XOR V = 1));

111 - desvio incondicional.

Deslocamento - é um valor sinalizado de dez *bits* que somado ao conteúdo atual do PC especifica o endereço para o qual o programa será desviado.

4.2. Modos de Endereçamento

Existem ao todo sete modos diferentes de acessar um dado com uma instrução *Assembly*. Qualquer um deles pode ser utilizado para acessar um operando utilizado como fonte de uma operação. Para acesso a um operando de destino, apenas quatro modos estão disponíveis, conforme podemos observar na tabela 4-2:

Fonte	Destino
Registrador	Registrador
Indexado	Indexado
Simbólico	Simbólico
Absoluto	Absoluto
Indireto	
Indireto com auto-incremento	
Imediato	

Tabela 4-2

Os modos de endereçamento válidos para operandos fonte são selecionados pelos campos As e S-Reg, no próprio *op-code* da instrução, conforme a tabela 4-3:

Campo As	Campo S-Reg	Modo
00	0000 a 1111	Registrador
01	0000	Simbólico (indexado ao PC)
01	0010	Absoluto (indexado ao CG1)
01	0001, 0011 a 1111	Indexado
10	0000 a 1111	Indireto
11	0001 a 1111	Indireto com auto-incremento
11	0000	Imediato

Tabela 4-3

Os modos de endereçamento válidos para os operandos de destino são selecionados pelo *bit* Ad e pelo campo D-Reg no *op-code* da instrução, conforme a tabela 4-4:

Campo Ad	Campo D-Reg	Modo
0	xxxx	Registrador
1	0000	Simbólico (indexado ao PC)
1	0010	Absoluto (indexado ao CG1)
1	0001, 0011 a 1111	Indexado

Tabela 4-4

4.2.1. Modo Imediato

Iniciaremos pelo modo imediato, pois é com ele que podemos inicializar um registrador ou posição de memória com um valor qualquer.

O modo imediato caracteriza-se pela presença de uma constante operando como fonte da operação. Em *Assembly*, a constante deve ser precedida do símbolo "#".

Note que esse modo somente pode ser utilizado para operandos fonte de uma operação.

Vejamos alguns exemplos utilizando a instrução MOV (copia um dado do operando fonte para o operando destino):

1. Para carregar o registrador R5 com o valor 0x1000:

```
MOV #0x1000,R5
```

A codificação da instrução anterior ocorre da seguinte forma: 0x35400010, ou seja, instrução = 0x4035 e primeiro operando = 0x1000. A codificação binária da instrução ocorre conforme em seguida:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	0	0	0	0	0	1	1	0	1	0	1

Tabela 4-5

Repare que a codificação do campo "As" indica que o operando fonte deve ser lido do registrador indicado por S-Reg, que indica como fonte o registrador R0 (PC).

Como após a leitura do primeiro *word* da instrução o PC está apontando para o primeiro operando dela, a CPU vai ler esse valor, que será escrito no local indicado pelo *bit* Ad e pelo campo D-Reg, no caso, o registrador número 5 ou R5.

Observe que o *bit* B/W está em nível 0, indicando que a operação será de 16 *bits*. Esta é a condição-padrão utilizada pelo montador. Alternativamente, podemos utilizar o sufixo ".W" para informar ao montador (*assembler*) que a operação será de 16 *bits*.

O resultado final após a execução da instrução será que o registrador R5 irá receber o valor 0x1000.

Vejamos outro exemplo utilizando uma operação de 8 *bits*:

2. Para carregar o registrador R5 com o valor 0x10:

```
MOV.B #0x10,R5
```

Repare a presença do sufixo ".B", que sinaliza a operação de 8 *bits*.

A codificação desta instrução será 0x75401000, ou seja: instrução = 0x4075 e operando = 0x0010. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	0	0	0	0	1	1	1	0	1	0	1

Tabela 4-6

Repare que agora o *bit* B/W está setado, indicando que a operação será de 8 *bits*.

Observe também que, apesar de ser uma operação de 8 *bits*, o operando deve estar em formato de 16 *bits*. Apenas a parte menos significativa do operando será escrita no registrador de destino. A parte mais significativa do registrador de destino será apagada.

Na prática, essa operação não difere da operação de 16 *bits*, mas isso só ocorre porque o destino é um registrador de 16 *bits*.

Se o destino fosse um endereço de memória de 8 *bits*, o efeito seria diferente:

3. Escrevendo o valor 0x29 no endereço 0x200 da memória:

```
MOV.B #0x29,&0x200
```

Repare que utilizamos o operador "&" para especificar que o segundo operando é um endereço de memória.

A codificação desta instrução na memória será 0xF24029000002, ou seja, instrução = 0x40F2, primeiro operando = 0x0029 e segundo operando = 0x0200. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	0	0	0	1	1	1	1	0	0	1	0

Tabela 4-7

Observe que este exemplo é similar ao anterior, porém, neste caso, o modo de endereçamento do destino é diferente. O *bit* Ad setado e o campo D-Reg em 2 (registrador R2) indicam que o destino é um endereço absoluto especificado pelo segundo operando, no caso 0x0200.

Após a execução da instrução, o valor 0x29 será escrito no endereço 0x200 de memória. O conteúdo do endereço seguinte (0x201) permanece inalterado.

O último exemplo do modo imediato demonstra a escrita de um valor imediato de 16 *bits* em uma posição da memória.

4. Escrevendo o valor 0x1234 no endereço 0x200 da memória:

```
MOV.W #0x1234,&0x200
```

A instrução apresentada será codificada na memória da seguinte forma: 0xB24034120002, ou seja, instrução = 0x40B2, primeiro operando = 0x1234 e segundo operando = 0x0200. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	0	0	0	1	0	1	1	0	0	1	0

Tabela 4-8

A única alteração em relação ao exemplo anterior é o *bit* B/W que está apagado, indicando uma operação de 16 *bits*.

Ao final da execução desta instrução, o conteúdo do endereço 0x0200 será igual a 0x34 e o conteúdo do endereço 0x0201 será igual a 0x12.

Observação: *A utilização de instruções de 8 bits não implica em utilização de operandos de 8 bits. Nos MSP430, os operandos possuem sempre um tamanho de 16 bits, independente de a instrução ser de 8 ou 16 bits. No caso de instruções de 8 bits, a parte alta do operando é sempre descartada.*

4.2.2. Modo Registrador

Neste modo, o operando fonte ou destino é um registrador da CPU.

Este modo é selecionado quando o campo As = 00 (a fonte é um registrador da CPU), ou Ad = 0 (o destino é um registrador da CPU).

O modo registrador pode ser utilizado tanto para operandos fonte quanto para operandos destino, ou ambos.

Vejamos alguns exemplos:

Primeiramente, vejamos como carregar um registrador com o valor de outro registrador:

1. Carregando R15 com o valor de R4:

```
MOV.W R4,R15
```

A codificação da instrução será 0x0F44, sendo a instrução 0x440F. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	1	0	0	0	0	0	0	1	1	1	1

Neste caso, podemos constatar que o campo As é igual a 00, indicando que o modo de endereçamento do operando fonte é o modo registrador. O campo S-Reg especifica o registrador de número 4 (0100) ou R4 como sendo a fonte da operação.

O *bit* Ad especifica que o destino é um registrador indicado pelo campo D-Reg, que contém o valor 15 (1111), ou seja, o registrador R15.

Após a execução da instrução, o registrador R15 recebe o conteúdo do registrador R4, permanecendo este último inalterado.

O segundo exemplo demonstra como carregar um endereço da memória com o valor de um registrador:

2. Carregando o endereço 0x200 com o valor de R15 numa operação de 8 *bits*:

```
MOV.B R15,&0x200
```

A codificação da instrução será 0xC24F0002, sendo a instrução = 0x4FC2 e o primeiro operando = 0x0200. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	1	1	1	1	1	1	0	0	0	0	1	0

Tabela 4-10

Novamente neste caso temos o campo As indicando que o operando fonte é o registrador 15 (pois S-Reg = 15).

O *bit* Ad = 1 indica que o destino é um endereço de memória especificado pelo primeiro operando (no caso o endereço 0x0200).

Como esta é uma operação de 8 *bits*, o endereço 0x200 recebe o conteúdo da parte baixa do registrador R15. Nenhuma outra posição de memória será afetada e o conteúdo do registrador R15 não é alterado.

O terceiro exemplo demonstra como carregar um registrador com o valor lido de um endereço de memória:

3. Lendo o conteúdo do endereço 0x200 e escrevendo em R5, em uma operação de 16 *bits*:

```
MOV.W &0x200,R5
```

A codificação da instrução será 0x15420002, sendo a instrução = 0x4215 e o primeiro operando = 0x0200. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	0	1	0	0	0	0	1	0	1	0	1

Tabela 4-11

O campo As = 01 indica que a instrução utiliza um modo de endereçamento indexado a um registrador (apontado por S-Reg). Como S-Reg = 2 (registrador R2), temos o modo de endereçamento absoluto para o operando fonte. O endereço absoluto é lido no primeiro operando (que é igual a 0x0200).

O campo Ad = 0 indica que o destino é um registrador especificado por D-Reg (no caso 5 ou R5).

Como a operação é de 16 *bits*, esta instrução copia o conteúdo do endereço 0x0200 para o *byte* menos significativo de R5 e o conteúdo do endereço 0x0201 para o *byte* mais significativo de R5.

4.2.3. Modo Indexado

O modo indexado permite que se utilize uma constante (endereço absoluto), que somada a um registrador vai compor o endereço do operando fonte ou do operando de destino.

Esse modo é particularmente útil no acesso a tabelas ou matrizes, em que o endereço do primeiro elemento da matriz é somado a um índice (o conteúdo do registrador) para formar o endereço do elemento desejado.

Para utilizar esse modo de endereçamento, o valor do endereço base deve ser fornecido, seguindo-o com o registrador desejado para o índice entre parênteses. Com exceção dos registradores PC (R0) e CG1(R2), qualquer outro registrador pode ser utilizado como índice no modo indexado. O PC e o CG1, quando utilizados como índice, originam os modos simbólico e absoluto, que serão vistos mais adiante.

O modo indexado pode ser utilizado tanto para operandos fonte quanto para operandos destino, inclusive simultaneamente.

Vejamos alguns exemplos de utilização desse modo de endereçamento:

O primeiro exemplo demonstra como carregar uma constante imediata em um endereço indexado.

1. Carregando o valor 0xAB no endereço 0x200 mais o índice formado por R9:

```
MOV.B #0xAB,0x200(R9)
```

A codificação da instrução na memória será 0xF940AB000002, em que instrução = 0x40F9, primeiro operando = 0x00AB e segundo operando = 0x0200. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	0	0	0	1	1	1	1	1	0	0	1

Tabela 4-12

O campo As = 11 sinaliza o operando fonte imediato que é obtido por meio do endereço apontado por S-Reg (que indica o PC). Como após a leitura do primeiro *word* da instrução o PC está apontando para o primeiro operando, ele será o valor lido (0x00AB).

O campo Ad indica que o destino é um endereço formado pela soma do segundo operando (0x0200) com o conteúdo do registrador determinado por D-Reg (9 ou R9).

Desta forma o *byte* lido do operando imediato (0xAB) será escrito no endereço 0x200+R9.

Outros exemplos do modo indexado:

Copiando o valor de 8 *bits* do elemento R8 da "tabela1" para o R5:

```
MOV.B    tabela1(R8),R5
```

Copiando o valor de 16 *bits* do registrador de captura do canal 0 do *timer* A para o elemento especificado pelo registrador R7 na "tabela2":

```
MOV.W    TACCR0, tabela2(R7)
```

Copiando um valor de 16 *bits* do elemento apontado por R5 em "tabela1" para o elemento apontado por R11 em "tabela2". Lembre-se de que os índices R5 e R11 das tabelas devem ter sempre conteúdo par, de forma a acessar endereços pares da memória, pois se trata de uma operação de 16 *bits*:

```
MOV.W    tabela1(R5),tabela2(R11)
```

4.2.4. Modo Simbólico

O modo simbólico é uma variação do modo indexado. Neste caso, o registrador de índice é o R0 (o contador de programa ou PC).

No modo simbólico o endereço do operando fonte ou de destino é obtido pela soma de uma constante imediata ao conteúdo do PC.

Vejamos um exemplo desse modo de endereçamento. Vamos copiar o conteúdo de um registrador para um endereço simbólico:

1. Copiando o conteúdo de R8 para o endereço representado pelo símbolo TESTE (localizado no endereço 0x0200 da memória) em uma operação de 8 *bits*:

```
MOV.B R8,TESTE
```

A codificação da instrução depende do endereço em que ela foi montada, já que para calcular o valor de deslocamento a ser adicionado ao registrador de índice (o PC), é necessário conhecer o endereço da instrução.

Supondo que ela esteja montada no endereço 0xF806, a sua codificação será 0xC048F809, em que instrução = 0x48C0 e o primeiro operando = 0x09F8. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	1	0	0	0	1	1	0	0	0	0	0	0

Tabela 4-13

Podemos observar que o campo As indica que o operando fonte é o registrador determinado por S-Reg (8 ou R8).

O *bit* Ad = 1 indica que o modo de endereçamento do destino é indexado e o campo D-Reg = 0 determina que a indexação será referente ao R0 ou PC.

Para calcular o endereço do operando de destino, o primeiro operando (0x09F8) é somado ao valor atual do PC (que após a leitura da instrução está apontando para o endereço 0xF808).

Neste caso, o endereço do destino será 0xF808 + 0x09F8 = 0x10200. Como somente os 16 *bits* LSB são utilizados, temos o endereço de destino igual a 0x0200.

Após a execução da instrução, o conteúdo do endereço 0x0200 será igual ao valor do *byte* menos significativo do registrador R8. Esse último registrador permanece inalterado.

Note que todo o processo de cálculo de deslocamentos na montagem da instrução é feito pelo *assembler* e é transparente para o usuário.

Outros exemplos do modo simbólico:

Copiando o valor de "TESTE" para o registrador R15:

```
MOV.B    TESTE,R15
```

Copiando o valor de 16 *bits* do registrador de captura do canal 1 do *timer* A para o elemento especificado pelo registrador R10 na "tabela2". O primeiro operando (TACCR1) é endereçado pelo modo simbólico.

```
MOV.W    TACCR1, tabela2(R10)
```

4.2.5. Modo Absoluto

A segunda variação do modo indexado é chamada de modo absoluto. Nesse modo, o endereço absoluto do operando fonte ou destino é informado diretamente como operando da instrução.

A diferença entre o modo absoluto e o modo indexado é que neste caso o registrador utilizado para indexar o endereço é o R2, atuando como gerador de constantes (CG1) e gerando a constante 0, ou seja, o endereço será o próprio valor do operando.

Para especificar um endereço absoluto, utilizamos o operador "&" em frente.

Nós já vimos alguns exemplos de utilização do modo absoluto, mas vejamos um novo exemplo, desta vez utilizando os modos simbólico e o absoluto:

1. Copiando o conteúdo do endereço especificado pelo símbolo TESTE (0x0200) para o endereço absoluto 0x208. Supondo que a instrução esteja montada no endereço 0xF80A:

```
MOV.B TESTE,&0x208
```

A codificação da instrução será 0xD240F4090802, sendo a instrução = 0x40D2, primeiro operando = 0x09F4 e o segundo operando = 0x208. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	0	0	0	1	1	0	1	0	0	1	0

Tabela 4-14

O campo As = 01 indica que o modo de endereçamento do operando fonte é o indexado e S-Reg = 0 indica que o registrador R0 (PC) será utilizado como índice, o que se traduz como o modo de endereçamento simbólico.

O endereço do operando fonte é calculado somando o operando 0x09F4 ao valor atual do registrador de índice, que é o PC (endereço da instrução + 2, ou 0xF80C): 0x09F4 + 0xF80C = 0x10200 ou 0x0200 em 16 *bits*.

O campo Ad = 1 indica que o modo indexado será também utilizado para o operando de destino. Neste caso, o registrador de índice é o R2, que nesse modo atua sempre como CG1, gerando a constante 0.

O endereço do operando de destino é calculado somando o segundo operando da instrução (0x0208) com o conteúdo do registrador de índice (CG1, que vale 0): 0x0208 + 0 = 0x0208.

Outros exemplos do modo absoluto:

Copiando o valor de "TESTE" para o registrador R15. A referência a "TESTE" é feita pelo seu endereço absoluto:

```
MOV.B    &TESTE,R15
```

Copiando o valor do registrador de captura do canal 1 do timer A para o elemento especificado pelo registrador R10 na "tabela2". O primeiro operando (TACCR1) é endereçado pelo modo absoluto.

```
MOV.W    &TACCR1, tabela2(R10)
```

Copiando o valor de "TESTE" para o "TEMP". A referência a "TESTE" é feita pelo seu endereço absoluto, enquanto "TEMP" é endereçada simbolicamente:

```
MOV.B    &TESTE,TEMP
```

4.2.6. Modo Indireto

No modo de endereçamento indireto, o registrador (qualquer um dos 16 da CPU) serve como apontador do endereço em que está localizado o operando. Ou seja, o conteúdo do registrador é o endereço do operando fonte.

Esse modo somente pode ser utilizado para o operando fonte. No caso do operando de destino, é possível utilizar como alternativa o modo indexado, utilizando o deslocamento igual a zero: 0(Rx), sendo Rx um dos registradores da CPU (R0 a R15).

Vejamos alguns exemplos dessa modalidade de endereçamento:

1. Supondo que R6 contenha o valor 0x208, podemos copiar o conteúdo do endereço 0x208 para o endereço absoluto 0x201 pela seguinte instrução:

```
MOV.B @R6,&0x201
```

A codificação da instrução será 0xE2460102, sendo a instrução = 0x46E2 e o primeiro operando = 0x0201. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	1	1	0	1	1	1	0	0	0	1	0

Tabela 4-15

O campo As = 10 indica o modo de endereçamento indireto e o campo S-Reg indica que o registrador fonte é o R6.

O operando fonte será lido do endereço apontado por R6, ou seja, 0x0208.

O campo Ad = 1 indica que o modo de endereçamento do destino é indexado e D-Reg = 2 seleciona o R2. Nesse modo o R2 atua como gerador de constantes (CG1), gerando a constante 0. Este é o modo absoluto.

O conteúdo do operando (0x201) é somado a 0 para formar o endereço do destino da operação (0x201).

Ao término da instrução, o conteúdo do endereço 0x0208 será copiado para o endereço 0x0201.

Vejamos agora um exemplo de acesso indireto tanto no operando fonte quanto no operando de destino:

2. Copiando o conteúdo da posição de memória apontada por R5 para a posição de memória apontada por R6:

Neste caso, não temos disponível o modo indireto para o operando de destino, mas podemos utilizar uma alternativa que é o modo indexado com endereço base igual a zero:

```
MOV.B @R5,0(R6)
```

A codificação da instrução na memória será 0xE6450000, sendo a instrução = 0x45E6 e o primeiro operando = 0x0000. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	1	0	1	1	1	1	0	0	1	1	0

Tabela 4-16

O campo As = 10 identifica o modo de endereçamento indireto para o operando fonte, utilizando o R5 (S-Reg = 0101) como apontador.

O campo Ad = 1 indica que o modo de endereçamento do operando de destino é o indexado e D-Reg = 6 identifica o registrador R6 que será utilizado como índice. O operando 0x0000 será lido e somado ao conteúdo de R6 para determinar o endereço do operando de destino.

Ao término da instrução, o conteúdo do endereço apontado por R5 será lido e escrito no endereço apontado por R6.

Outros exemplos do modo indireto:

Copiando o valor de 16 *bits* do endereço apontado por R14 para a variável "TESTE". A referência a "TESTE" é feita pelo seu endereço absoluto (lembre-se de que em operações de 16 *bits*, somente é possível acessar endereços pares da memória):

```
MOV.W    @R15, &TESTE
```

Copiando o valor de 8 *bits* da posição de memória apontada por R8 para o elemento especificado pelo registrador R9 na "tabela2".

```
MOV.B    @R8, tabela2(R9)
```

4.2.7. Modo Indireto com Auto-Incremento

Esse modo é similar ao anterior, mas com a diferença de que o conteúdo do registrador utilizado como apontador é automaticamente incrementado após a sua leitura. Nesse modo de endereçamento, somente podem ser utilizados os registradores R1 a R15. O PC (R0) não pode ser utilizado nessa modalidade de endereçamento.

Dependendo do tipo de acesso utilizado na instrução (*byte* ou *word*), o registrador apontador é incrementado em 1 (para operações *byte*) ou em 2 (para operações *word*).

Observação: *O comportamento do registrador R1 (SP) nesse modo é bastante peculiar. Quando utilizado com a instrução MOV, o seu conteúdo é sempre incrementado em 2, independente de a instrução ser de 8 ou 16 bits. Quando utilizado com outras instruções, o seu conteúdo somente é incrementado em 2 nas instruções de 16 bits. No caso de instruções de 8 bits, não ocorre auto-incremento.*

Esse modo de endereçamento somente está disponível para operandos fonte. Para operandos destino é possível utilizar o modo indexado (com índice zero), seguido de uma operação de incremento do registrador.

Vejamos alguns exemplos de utilização desse modo de endereçamento:

1. Lendo um valor de 8 *bits* de um endereço apontado por R10, utilizando o auto-incremento e guardando o resultado em R4 (supondo que R10=0x200 e o conteúdo desse endereço seja igual a 0x89):

```
MOV.B @R10+,R4
```

A codificação da instrução na memória será 0x744A, sendo a instrução = 0x4A74. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	1	0	1	0	0	1	1	1	0	1	0	0

Tabela 4-17

Os campos As = 11 e S-Reg = 10 indicam que o modo de endereçamento da fonte é o indireto com auto-incremento, sendo R10 o apontador do endereço do operando.

O campo Ad = 0 indica que o destino é o registrador especificado por D-Reg, no caso o registrador 4 ou R4.

O campo B/W em 1 indica uma operação de 8 *bits*.

Após a execução dessa instrução, o registrador R4 conterá o valor 0x0089 e o registrador R10 conterá o valor 0x201.

Um outro exemplo, desta vez demonstrando a utilização desse modo na operação de retirada de informação da pilha (POP):

2. Supondo que o registrador SP esteja apontando para o endereço 0x210 e este endereço contenha o valor 0xAB, a seguinte instrução poderia ser utilizada para realizar a operação de retirada do valor do topo da pilha (POP) e o seu armazenamento no endereço absoluto 0x200:

```
MOV.B @SP+,&0x200
```

A codificação da instrução na memória será 0xF2410002, sendo a instrução = 0x41F2 e operando = 0x0200. A codificação binária da instrução será:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	0	0	1	1	1	1	1	0	0	1	0

Tabela 4-18

Os campos As = 11 e S-Reg = 1 indicam que o modo de endereçamento é o indireto com auto-incremento do SP.

O campo Ad = 1 indica que o modo de endereçamento do destino é indexado ao CG1 (que neste caso é sempre igual a zero), sendo o endereço do destino calculado pela soma do operando que segue a instrução (0x0200) mais 0 (do CG1).

Desta forma concluímos que a instrução vai ler o conteúdo da posição de memória indicada por SP, somar dois ao seu conteúdo (já que essa instrução envolvendo o SP neste modo de endereçamento provoca o incremento dele em 2) e em seguida escrever o valor lido na posição de memória indicada (endereço 0x0200).

Esse tipo de operação será visto em maiores detalhes quando estudarmos a instrução POP.

4.3. Instruções Físicas e Instruções Emuladas

Uma característica-chave na arquitetura MSP430 é o seu conjunto de instruções muito reduzido: existem apenas 27 diferentes instruções reconhecíveis pela CPU.

A adoção de um pequeno conjunto de instruções implica em menor quantidade de *bits* para formar um *op-code*, o que reduz a quantidade de memória necessária para armazenar uma instrução.

Por outro lado, um pequeno conjunto de instruções significa que o programador vai precisar, em muitos casos, de mais de uma instrução para a execução de uma operação não atendida pelas instruções existentes.

Um exemplo claro dessa implicação pode ser encontrado na instrução de apagamento de um registrador ou posição da memória. Na maioria das arquiteturas existentes, o leitor encontra uma instrução específica para realizar essa operação, normalmente reconhecida pelo mnemônico CLR (do inglês ***CLeaR***, ou apagar).

Nos MSP430, essa instrução não existe entre os *op-codes* físicos (as 27 instruções reconhecidas pela CPU). Mas como então se realiza o apagamento de um registrador ou posição de memória?

A resposta pode ser encontrada na própria natureza da operação de apagamento: ela consiste em uma movimentação de dados, na qual se copia o valor 0 (zero) para o destino especificado. Os MSP430 possuem uma instrução de movimentação de dados, chamada MOV e que já comentamos anteriormente.

Sendo assim, a seguinte instrução deve promover o apagamento do destino especificado, tal como uma instrução CLR o faria:

```
MOV #0,destino
```

Pronto, resolvida a questão!

Na verdade não. Uma instrução de dois operandos como `MOV #0,destino` acabaria ocupando mais memória, sendo, portanto, mais lenta que uma instrução `CLR destino` que possui somente um operando.

Para resolver problemas como o descrito, foram criados os registradores geradores de constantes (CG1 e CG2).

Esses registradores podem atuar como operandos fonte em instruções que necessitem de uma das seis constantes predefinidas pelo fabricante.

A escolha entre uma das constantes é feita de acordo com o modo de endereçamento em que se acessa o registrador (CG1 ou CG2).

Registrador	Campo As	Constante	Significado
R2 (CG1)	00	nenhuma	Acesso ao registrador R2
R2 (CG1)	01	(0)	Modo de endereçamento absoluto
R2 (CG1)	10	0x0004	Constante 4, útil para acesso ao *bit* N no SR
R2 (CG1)	11	0x0008	Constante 8, útil para acesso ao *bit* GIE no SR
R3(CG2)	00	0x0000	Constante 0, útil para operações de apagamento
R3(CG2)	01	0x0001	Constante 1, útil para acesso ao *bit* C no SR
R3(CG2)	10	0x0002	Constante 2, útil para acesso ao *bit* Z no SR
R3(CG2)	11	0xFFFF	-1, útil para operação de inversão de *bits*

Tabela 4-19

Assim, uma instrução para apagamento do registrador R9 é escrita na memória do microcontrolador como:

```
MOV #0,R9
```

E gera o seguinte código na memória (graças ao uso dos geradores de constantes): 0x0943, sendo a instrução igual a 0x4309. O código binário da instrução é:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			
0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	1

Tabela 4-20

Observe que o operando fonte é indicado como o registrador R3 (As = 00 e S-Reg = 3).

Se observarmos a tabela 4-19, podemos perceber que quando o registrador R3 é acessado como operando fonte e tendo As = 00, ele gera a constante 0x0000 e que é copiada para o registrador destino (R9), fazendo com que o seu conteúdo seja apagado.

É fácil perceber a vantagem da utilização dos geradores de constantes. O operando imediato (no caso a constante 0) não precisou ser fornecido como um operando físico da instrução, resultando em uma instrução com tamanho menor, portanto mais rápida.

Existe um total de 24 operações que são emuladas pelo MSP430, sem nenhum ônus de performance ou de tamanho de código e apesar dessas operações não possuírem *op-codes* próprios, possuem mnemônicos próprios. Isso ocorre para facilitar a legibilidade do código, ao mesmo tempo em que aproxima a linguagem *Assembly* desses *chips* das outras existentes no mercado.

4.4. Conjunto de Instruções

Para fins didáticos, dividimos o conjunto de 51 instruções do MSP430 em algumas categorias, conforme a afinidade de operação:

- **Instruções de movimentação e manipulação de dados** - utilizadas para tarefas de carga de valores em memória ou em registradores, além de movimentação de dados entre registradores e/ou posições de memória:
  - MOV - copia dados da fonte para o destino;
  - CLR - apaga o conteúdo do destino;
  - SWPB - troca os *bytes* do destino;
  - PUSH - armazena um dado na pilha;
  - POP - restaura um dado da pilha;
  - BIC - apaga um ou mais *bits* do destino;
  - BIS - seta um ou mais *bits* do destino.
- **Instruções aritméticas e lógicas** - utilizadas para realizar operações matemáticas e lógicas. Incluiremos nesta categoria as instruções de incremento e decremento:
  - ADD - adição;
  - ADDC - adição com o transporte;
  - ADC - adição do transporte;
  - DADD - adição decimal;
  - DADC - adição decimal do transporte;
  - SUB - subtração;
  - SUBC - subtração com empréstimo;
  - SBC - subtrai o *bit* de empréstimo;
  - INC - incremento;
  - INCD - incrementa em dois;
  - DEC - decremento;
  - DECD - decremento de dois;
  - RLA - rotação aritmética à esquerda;
  - RRA - rotação aritmética à direita;
  - SXT - extensão do sinal;
  - AND - operação lógica E;
  - XOR - operação lógica OU exclusivo;
  - INV - inverte os *bits*;
  - RLC - rotação à esquerda por meio do transporte;
  - RRC - rotação à direita por meio do transporte.

- **Instruções de teste e desvio** - utilizadas para testes condicionais e desvios do fluxo do programa (desvios relativos, absolutos e chamadas de sub-rotina):
  - BIT - teste de *bits*;
  - CMP - comparação;
  - TST - testa se igual a zero;
  - BR - desvio absoluto incondicional;
  - JMP - salto absoluto incondicional;
  - JEQ/JZ - desvia se igual ou zero;
  - JNE/JNZ - desvia se diferente ou não zero;
  - JC/JHS - desvia se C=1 ou se maior/igual;
  - JNC/JLO - desvia se C=0 ou se menor;
  - JGE - desvia se maior ou igual;
  - JL - desvia se menor;
  - JN - desvia se negativo;
  - CALL - chamada de sub-rotina;
  - RET - retorno de sub-rotina;
  - RETI - retorno de interrupção.
- **Instruções de controle da CPU** - utilizadas para controle do estado da CPU (especialmente *flags* do processador):
  - CLRC - apaga C;
  - CLRN - apaga N;
  - CLRZ - apaga Z;
  - SETC - seta C;
  - SETN - seta N;
  - SETZ - seta Z;
  - DINT - desabilita interrupções;
  - EINT - habilita interrupções;
  - NOP - nenhuma operação.

4.4.1. Instruções de Movimentação e Manipulação de Dados

Neste tópico vamos conhecer as instruções de manipulação e movimentação de dados.

Esse tipo de instrução é dos mais importantes em um dispositivo programável, já que qualquer programa necessita trabalhar e manipular dados.

Graças à ortogonalidade do conjunto de instruções, essas operações são feitas com grande facilidade e versatilidade.

Nesta categoria de instruções também incluiremos as instruções de manipulação de *bit*.

4.4.1.1. MOV — Copia um dado da fonte para o destino

Formato: MOV fonte,destino ou MOV.W fonte,destino

MOV.B fonte,destino

A instrução MOV é utilizada para copiar um dado de 8 ou 16 *bits* do local indicado pelo operando "fonte" para o local indicado pelo operando "destino".

O mnemônico MOV.W ou simplesmente MOV é utilizado para realizar uma operação de 16 *bits* (*word*), enquanto o mnemônico MOV.B é utilizado para uma operação de 8 *bits* (*byte*).

Op-code: (instrução com 2 operandos): 4 *bits* - 0100.

***Flags* afetados no SR:** nenhum.

Exemplos:			*Flags* alterados V	N	Z	C
`MOV.B`	`#10,R4`	`;copia o valor 10 decimal para R4 (R4=10).`	?	?	?	?
`MOV`	`R4,&0x200`	`;copia o valor de 16 bits de R4 para o endereço 0x200 (o conteúdo do endereço 0x200 passa a ser 10 e do endereço 0x201 passa a ser 0).`	?	?	?	?
`MOV.W`	`#0x1234,R5`	`;copia o valor 0x1234 para R5 (R5 = 0x1234).`	?	?	?	?
`MOV.B`	`R5,R6`	`;copia o valor de 8 bits de R5 (0x34) para R6 (R6 = 0x0034).`	?	?	?	?
`MOV.B`	`#0xAB,R5`	`;copia o valor de 8 bits 0xAB para R5. Observe que os 8 bits mais significativos de R5 são apagados em decorrência da operação (R5 = 0x00AB).`	?	?	?	?

4.4.1.2. CLR — Apaga o conteúdo do destino

Formato: CLR destino ou CLR.W destino

CLR.B destino

A instrução CLR é emulada com o uso da instrução MOV #0,destino. Após a sua execução, o conteúdo do operando indicado por "destino" é apagado.

O mnemônico CLR.W ou simplesmente CLR é utilizado para realizar uma operação de 16 *bits* (word) enquanto o mnemônico CLR.B é utilizado para uma operação de 8 *bits* (*byte*).

Op-code: emulada pela instrução MOV.

***Flags* afetados no SR:** nenhum.

Exemplos:			*Flags* alterados V	N	Z	C
`MOV.B`	`#0x1234,R9`	`;copia o valor 0x1234 para R9.`	?	?	?	?
`CLR.B`	`R9`	`;apaga os 16 bits de R9 (R9=0). Note que como o destino é um registrador de 16 bits, a operação de escrita de 8 bits provoca o apagamento dos 8 bits mais significativos.`	?	?	?	?
`MOV`	`#0x1234,&0x200`	`;copia em uma operação de 16 bits o valor 0x12 para o endereço 0x201 e o valor 0x34 para o endereço 0x200.`	?	?	?	?
`CLR.B`	`&0x201`	`;apaga o byte do endereço 0x201.`	?	?	?	?
`CLR`	`&0x200`	`;apaga em uma operação de 16 bits o conteúdo do endereço 0x200 e 0x201.`	?	?	?	?

4.4.1.3. SWPB — Troca os *bytes* do destino

Formato: SWPB destino

Essa instrução troca o conteúdo dos *bytes* do destino: o *byte* LSB recebe o conteúdo do *byte* MSB e vice-versa. A operação é sempre de 16 *bits*.

Op-code: (Instrução com 1 operando): 9 *bits* - 000100001.

***Flags* afetados no SR:** nenhum.

Exemplos:

			Flags alterados V	N	Z	C
`MOV.W`	`#0x1234,&0x200`	`;copia em uma operação de 16 bits o valor 0x34 para o endereço 0x200 e o valor 0x12 para o endereço 0x201.`	?	?	?	?
`SWPB`	`&0x200`	`;troca os bytes do word no endereço 0x200: o endereço 0x200 passa a conter 0x12 e o endereço 0x201 passa a conter o valor 0x34.`	?	?	?	?
`MOV`	`&0x200,R5`	`;copia em uma operação de 16 bits o conteúdo do endereço 0x200 para o registrador R5 (R5 = 0x3412).`	?	?	?	?
`SWPB`	`R5`	`;troca os bytes de R5 (R5 = 0x1234).`	?	?	?	?

4.4.1.4. PUSH — Armazena um dado na pilha

Formato: PUSH fonte ou PUSH.W fonte

PUSH.B fonte

Essa instrução armazena o conteúdo indicado pelo operando "fonte" no topo da pilha. Antes da operação de empilhamento, o conteúdo do SP é decrementado em 2, independentemente de a operação ser de 8 *bits* (PUSH.B) ou de 16 *bits* (PUSH ou PUSH.W).

Op-code: (instrução com 1 operando): 9 *bits* - 000100100.

***Flags* afetados no SR:** nenhum.

Exemplos:

			Flags alterados V	N	Z	C
`MOV`	`#0x400,SP`	`;carrega o valor 0x400 em SP, inicializando o topo da pilha neste endereço. Estado da pilha:`	?	?	?	?

SP →

Endereço:	Conteúdo
0x0400	????
0x03FE	????
0x03FC	????

`MOV`	`#0xABCD,R4`	`;copia o valor 0xABCD para o registrador R4.`	?	?	?	?

```
PUSH    R4          ;armazena o valor de 16 bits de R4 na   ?  ?  ?  ?
                    pilha. O SP passa a apontar para o
                    novo topo da pilha (em 0x03FE).
                    Estado da pilha:
```

SP →

Endereço:	Conteúdo
0x0400	????
0x03FE	0xABCD
0x03FC	????

```
PUSH.B  R4          ;armazena o valor de 8 bits de R4 na    ?  ?  ?  ?
                    pilha. O SP passa a apontar para o
                    novo topo da pilha (em 0x03FC).
                    Observe que, apesar de somente ser
                    armazenado um byte na pilha, o SP é
                    decrementado em dois. O conteúdo do
                    endereço 0x03FD é desconhecido.
                    Estado da pilha:
```

SP →

Endereço:	Conteúdo
0x0400	????
0x03FE	0xABCD
0x03FC	0x??CD

Um outro exemplo interessante decorre do armazenamento do próprio SP na pilha, considerando que a pilha continua no último estado da instrução anterior:

```
PUSH    SP          ;armazena o conteúdo do SP na pilha.    ?  ?  ?  ?
                    O valor efetivamente armazenado na
                    pilha já é o SP+2 (0x03FA) e não o
                    valor do SP antes da instrução
                    (0x03FC). Estado da pilha:
```

SP →

Endereço:	Conteúdo
0x0400	????
0x03FE	0xABCD
0x03FC	0x??CD
0x03FA	0x03FA

4.4.1.5. POP

Retira um dado da pilha

Formato: POP destino ou POP.W destino
POP.B destino

Essa instrução é emulada pela instrução MOV @SP+,destino e a sua utilidade é retirar (ler) um valor armazenado na pilha e apontado pelo registrador SP, armazenando esse valor no local indicado pelo operando "destino".

O registrador SP é incrementado em dois durante a operação (antes da escrita no destino especificado).

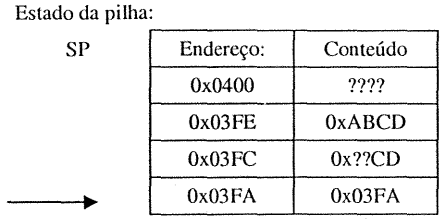
Op-code: emulada pela instrução MOV.

Exemplos:

Flags alterados

V N Z C

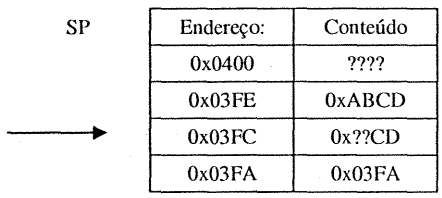
Estado da pilha:



Endereço:	Conteúdo
0x0400	????
0x03FE	0xABCD
0x03FC	0x??CD
0x03FA	0x03FA

```
POP     R9          ;retira o valor armazenado no topo da pilha e armazena em R9. Após a operação, R9 = 0x03FA. Estado da pilha:
```

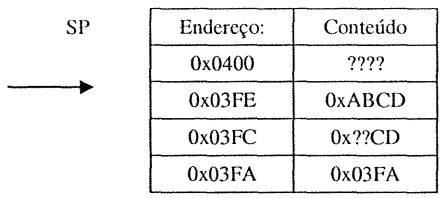
? ? ? ?



Endereço:	Conteúdo
0x0400	????
0x03FE	0xABCD
0x03FC	0x??CD
0x03FA	0x03FA

```
POP.B   R10         ;retira o valor armazenado no topo da pilha e armazena em R10. Após a operação, R10 = 0x00CD (byte MSB em 0, pois a operação foi de 8 bits). Estado da pilha:
```

? ? ? ?

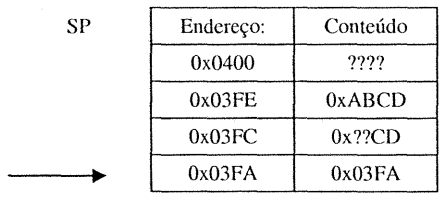


Endereço:	Conteúdo
0x0400	????
0x03FE	0xABCD
0x03FC	0x??CD
0x03FA	0x03FA

Um fato interessante ocorre quando tentamos retirar da pilha um valor do SP armazenado previamente. Supondo que a pilha esteja como no final do último exemplo da instrução PUSH:

```
POP     SP          ;retira o valor armazenado no topo da pilha e armazena em SP. Ao término da operação, SP = 0x03FA. Neste caso, o SP não foi alterado porque o incremento do SP ocorre antes da escrita final nele. Estado da pilha:
```

? ? ? ?



Endereço:	Conteúdo
0x0400	????
0x03FE	0xABCD
0x03FC	0x??CD
0x03FA	0x03FA

4.4.1.6. BIC — Apaga um ou mais *bits* no destino

Formato: BIC fonte,destino ou BIC.W fonte,destino
BIC.B fonte,destino

Essa instrução pode ser utilizada para realizar o apagamento de um ou mais *bits* do conteúdo de "destino".

Para realizar essa operação, o conteúdo de "fonte" é invertido logicamente e em seguida é feita a operação lógica E com o conteúdo de "destino".

Todos os *bits* que estiverem setados no operando fonte serão apagados no operando de destino.

Essa instrução pode operar tanto em 16 *bits* (BIC ou BIC.W) quanto em 8 *bits* (BIC.B).

No caso de utilização de *flags* criados pelo programador, deve-se procurar armazená-los nos *bits* 0,1,2 ou 3, de forma que o montador possa utilizar as constantes internas (1,2,4 ou 8) como operandos, diminuindo o tamanho da instrução e aumentando a velocidade do programa.

Op-code: (instrução com 2 operandos): 4 *bits* - 1100.

***Flags* afetados no SR:** nenhum.

Exemplos:

			Flags alterados			
			V	**N**	**Z**	**C**
`MOV.W`	`#0xFFFF, &0x200`	`;copia o valor 0xFFFF para o endereço 0x200 (tanto o endereço 0x200 quanto 0x201 recebem o valor 0xFF).`	?	?	?	?
`BIC.B`	`#0x81,&0x200`	`;apaga o bit 7 e o bit 0 do conteúdo do endereço 0x200 (que passa a conter o valor 0x7E).`	?	?	?	?
`BIC`	`#0x0E0F,&0x200`	`;apaga os bits 11,10,9,3,2,1 e 0 do conteúdo de 16 bits do endereço 0x200. O conteúdo do endereço 0x200 passa a ser 0x70 e o do endereço 0x201 passa a ser 0xF1.`	?	?	?	?

4.4.1.7. BIS — Seta um ou mais *bits* no destino

Formato: BIS fonte,destino ou BIS.W fonte,destino
BIS.B fonte,destino

Essa instrução permite setar um ou mais *bits* no destino especificado.

A CPU realiza uma operação lógica OU do conteúdo do operando "fonte" com o conteúdo do operando "destino", escrevendo o resultado em "destino".

Todos os *bits* que estiverem setados no operando fonte serão setados no operando de destino.

Essa instrução pode operar tanto em 16 *bits* (BIS ou BIS.W) quanto em 8 *bits* (BIS.B).

No caso de utilização de *flags* criados pelo programador, deve-se procurar armazená-los nos *bits* 0,1,2 ou 3, de forma que o montador possa utilizar as constantes internas (1,2,4 ou 8) como operandos, diminuindo o tamanho da instrução e aumentando a velocidade do programa.

Op-code: (instrução com 2 operandos): 4 *bits* - 1101.

***Flags* afetados no SR:** nenhum.

Exemplos:			*Flags* alterados V	N	Z	C
CLR.W	&0x200	;apaga o conteúdo dos endereços 0x200 e 0x201.	?	?	?	?
BIS.B	#1,&0x200	;seta o bit 0 do conteúdo do endereço 0x200. Esta instrução ocupa apenas 2 words de memória, pois utiliza uma constante do CG2.	?	?	?	?
BIS.W	#0xF00F,&0x200	;seta os bits 15,14,13,12,3,2,1 e 0 do conteúdo do endereço 0x200 e 0x201 (o valor de 16 bits na posição 0x200 passa a ser 0xF10F.	?	?	?	?

4.4.2. Instruções Aritméticas e Lógicas

Neste tópico, vamos estudar as instruções dedicadas às operações aritméticas e lógicas. Essas instruções fazem uso da unidade lógica e aritmética (ULA) que nos MSP430 é capaz de executar as seguintes operações:

Matemáticas:

- Adição e subtração binária;
- Adição decimal (BCD);
- Incremento e decremento simples;
- Incremento e decremento duplo;
- Extensão do sinal;
- Rotação aritmética à esquerda (multiplicação por 2) e à direita (divisão por 2).

Lógicas:

- E e OU Exclusivo *bit* a *bit*;
- Complemento de um (inversão dos *bits*);
- Rotação pelo *carry* à esquerda e à direita;

Repare que não há um mnemônico específico para realizar a operação OU. Neste caso, podemos utilizar a instrução BIS, que tem a mesma função.

Alguns modelos de MSP430 possuem uma unidade de multiplicação interna, mas ela não faz parte da ULA, sendo tratada, para todos os efeitos, como um periférico mapeado na memória. Maiores detalhes sobre esse módulo serão vistos no tópico 5.18.

4.4.2.1. ADD

Adiciona dois valores

Formato: ADD fonte,destino ou ADD.W fonte,destino
ADD.B fonte,destino

Essa instrução realiza a adição do conteúdo do operando "fonte" com o conteúdo do operando "destino", armazenando o resultado em "destino".

Observe que uma instrução com sufixo B realiza apenas a adição de 8 *bits* (o resultado escrito no destino terá sempre o *byte* mais significativo igual a zero), enquanto uma instrução sem sufixo, ou utilizando o sufixo .W, realiza a adição de 16 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com 2 operandos): 4 *bits* - 0101.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o resultado da operação envolvendo operandos sinalizados ultrapasse os limites de representação (menor que –32.768 ou maior que +32.767 para operações de 16 *bits* e menor que –128 ou maior que +127 para operações de 8 *bits*);

N - setado se o resultado da operação envolvendo operandos sinalizados for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso ocorra um transporte do *bit* 15 (operação de 16 *bits*) ou do *bit* 7 (operação de 8 *bits*), indicando que o resultado é maior que 65.535 (16 *bits*) ou 255 (8 *bits*).

			***Flags* alterados**			
Exemplos:			**V**	**N**	**Z**	**C**
MOV	#127,R5	;copia o valor 127 para R5.	?	?	?	?
ADD.B	#3,R5	;adiciona 3 ao conteúdo de R5 (R5 = 130). O flag V é setado pois o resultado é maior que +127 e o flag N é setado porque o bit 7 do resultado terminou setado. Z =0 pois o resultado de 8 bits foi diferente de zero e C = 0 pois não houve transporte do bit 7 para o bit 8.	1	1	0	0
MOV.B	#-1,R6	;copia o valor -1 para o registrador R6 (R6 = 0xFF). Observe que os flags não são alterados pela instrução e mantêm o seu estado anterior.	1	1	0	0
ADD.B	#-100,R6	;adiciona -100 (0x9C) ao conteúdo de R6 (R6 = 0x9B ou -101 decimal). O flag N = 1 pois o resultado é negativo (bit 7 do resultado igual a 1), o flag C = 1 porque houve transporte do bit 7 para o bit 8 (a soma de 0x9C + 0xFF resultou maior que 0xFF). V = 0 pois o resultado da operação não ultrapassou os limites de um valor de 8 bits sinalizado. Z = 0 porque o resultado é diferente de zero.	0	1	0	1
MOV	#0xFFFF,R7	;copia o valor 0xFFFF para R7	0	1	0	1
ADD.B	#1,R7	;adiciona o valor 1 ao conteúdo de R7. Como esta é uma operação de 8 bits, a soma será 1 + 255 (0xFF) = 256 (0x100), o que significa que o resultado de 8 bits será igual a zero (R7 = 0). Neste caso, o flag Z = 1 indicando o resultado igual a zero e C = 1 indicando o transporte de 8 bits (do bit 7 para o bit 8).	0	0	1	1
MOV	#0xFFFF,R8	;copia o valor 0xFFFF para R8	0	0	1	1
ADD	#1,R8	;adiciona 1 ao conteúdo de R8. Neste caso a operação será: 1 + 0xFFFF = 0x10000, indicando que o resultado de 16 bits será igual a zero. O flag Z = 1 indicando o resultado igual a zero e C = 1 indicando o transporte de 16 bits (do bit 15 para o C).	0	0	1	1

4.4.2.2. ADDC — Adiciona o transporte e o fonte ao destino

Formato: ADDC fonte,destino ou ADDC.W fonte,destino
ADDC.B fonte,destino

Essa instrução adiciona o conteúdo do *flag* C ao conteúdo de "fonte" mais o conteúdo de "destino", armazenando o resultado em "destino".

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com 2 operandos): 4 *bits* - 0110.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o resultado da operação envolvendo operandos sinalizados ultrapasse os limites de representação (menor que –32.768 ou maior que +32.767 para operações de 16 *bits* e menor que –128 ou maior que +127 para operações de 8 *bits*);

N - setado se o resultado da operação envolvendo operandos sinalizados for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso ocorra um transporte do *bit* 15 (operação de 16 *bits*) ou do *bit* 7 (operação de 8 *bits*), indicando que o resultado é maior que 65.535 (16 *bits*) ou 255 (8 *bits*).

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV`	`#500,R5`	`;copia o valor 500 decimal para R5 (R5 = 500).`	?	?	?	?
`BIS`	`#1,SR`	`;seta o bit 1 (carry) no SR.`	?	?	?	1
`ADDC`	`#100,R5`	`;soma o C mais o operando imediato (100 decimal) ao conteúdo de R5 (R5 = 100 + 1 + 500 = 601).`	0	0	0	0

4.4.2.3. ADC — Adiciona o transporte ao destino

Formato: ADC destino ou ADC.W destino
ADC.B destino

Essa instrução realiza a adição do *bit* C (transporte) ao conteúdo do destino especificado. A operação é emulada pela instrução ADDC #0,destino.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: emulada pela instrução ADDC.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o resultado da operação envolvendo operandos sinalizados ultrapasse os limites de representação (menor que –32.768 ou maior que +32.767 para operações de 16 *bits* e menor que –128 ou maior que +127 para operações de 8 *bits*);

N - setado se o resultado da operação envolvendo operandos sinalizados for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso ocorra um transporte do *bit* 15 (operação de 16 *bits*) ou do *bit* 7 (operação de 8 *bits*), indicando que o resultado é maior que 65.535 (16 *bits*) ou 255 (8 *bits*).

Exemplos:			*Flags* alterados V	N	Z	C
MOV	#10,R5	;copia o valor 10 decimal para R5 (R5 = 10).	?	?	?	?
BIS	#1,SR	;seta o bit 1 (carry) no SR.	?	?	?	1
ADC.W	R5	;soma o C ao R5 (R5 = 10 + 1 = 11).	0	0	0	0

4.4.2.4. DADD — Adiciona em decimal o transporte e o fonte ao destino

Formato: DADD fonte,destino ou DADD.W fonte,destino
DADD.B fonte,destino

Essa instrução adiciona em decimal o conteúdo do *flag* C ao conteúdo de "fonte" mais o conteúdo de "destino", armazenando o resultado em "destino".

Por se tratar de uma adição decimal, os operandos são considerados codificados em BCD de quatro dígitos. Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com 2 operandos): 4 *bits* - 1010.

***Flags* afetados no SR:** V, N, Z e C.

V - não definido;

N - setado se o *bit* mais significativo do resultado estiver em "1";

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso o resultado seja maior que 99 (operação de 8 *bits*) ou 9.999 (operação de 16 *bits*).

Exemplos:			*Flags* alterados V	N	Z	C
MOV.B	#0x89,R5	;copia o valor 0x89 para R5 (R5 = 0x89).	?	?	?	?
BIC	#1,SR	;apaga o bit 1 (carry) no SR.	?	?	?	0
DADD.B	#0x15,R5	;soma em decimal o conteúdo de C mais o operando imediato (0x15) mais o conteúdo de R5 (0x89). Como se trata de uma operação de 8 bits e o resultado (0x104) é maior que 0x99, o registrador R5 recebe o valor 0x04 e o flag C é setado.	0	0	0	1

Em seguida temos um exemplo de uma soma decimal de dois valores de 32 *bits* BCD (o primeiro localizado em R4 e R5, o segundo em R6 e R7). O resultado é armazenado em R4 e R5:

MOV	#0x0001,R4		?	?	?	?
MOV	#0x0999,R5	; o primeiro número é igual a 10999.	?	?	?	?
CLR	R6		?	?	?	?
MOV	#0x9325,R7	; o segundo número é igual a 9325.	?	?	?	?
BIC	#1,SR	;apaga o carry no SR (C=0).	?	?	?	0

```
DADD    R7,R5     ;soma em decimal o conteúdo de R7 mais o C mais R5. O resultado de 16 bits (0x9325 + 0 + 0x0999 = 0x10324) é armazenado em R5 (R5 = 0x0324). Observe que o flag C foi setado, pois o resultado da operação foi maior que 9.999.
```

V	N	Z	C
0	0	0	1

```
DADD    R6,R4     ;soma em decimal o conteúdo de R6 mais o C mais R4. O resultado de 16 bits (0 + 1 + 1 = 0x0002) é armazenado em R4 (R4 = 0x0002). O flag C é apagado, pois o resultado foi menor que 9999.
```

V	N	Z	C
0	0	0	0

Ao final do programa anterior, R4 = 0x0002 e R5 = 0x0324, o que corresponde ao valor 0x00020324 BCD, ou seja, 20.324 decimal.

4.4.2.5. DADC



Adiciona em decimal o transporte ao destino

Formato: DADC destino ou DADC.W destino
DADC.B destino

Essa instrução realiza a adição decimal do *bit* C (transporte) ao conteúdo do destino especificado. A operação é emulada pela instrução DADD #0,destino.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: emulada pela instrução ADDC.

***Flags* afetados no SR:** V, N, Z e C.

V - não definido;

N - setado se o *bit* mais significativo do resultado estiver em "1";

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso o resultado seja maior que 99 (operação de 8 *bits*) ou 9.999 (operação de 16 *bits*).

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV`	`#0x89,R5`	`;copia o valor 0x89 (89 BCD) para o registrador R5.`	?	?	?	?
`BIS`	`#1,SR`	`;seta o bit 1 (carry) no SR.`	?	?	?	1
`DADC.B`	`R5`	`;soma decimal do C com R5 (R5 = 89 + 1 = 90). Ao final da operação, R5 = 0x0090`	0	0	0	0

4.4.2.6. SUB

Subtrai a fonte do destino

Formato: SUB fonte,destino ou SUB.W fonte,destino
SUB.B fonte,destino

Essa instrução realiza a subtração do conteúdo do operando "fonte" do conteúdo do operando "destino", armazenando o resultado em "destino".

A operação de subtração é realizada pela adição do conteúdo do operando "destino" com o complemento de dois do operando "fonte", ou seja, "destino" = "destino" + NOT "fonte" + 1.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com 2 operandos): 4 *bits* - 1000.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o resultado da operação envolvendo operandos sinalizados ultrapasse os limites de representação (menor que –32.768 ou maior que +32.767 para operações de 16 *bits* e menor que –128 ou maior que +127 para operações de 8 *bits*);

N - setado se o resultado da operação for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso não tenha sido necessário um empréstimo, ou seja, resultado positivo. Apagado caso tenha sido necessário um empréstimo (resultado negativo).

Exemplos:			***Flags* alterados** V	N	Z	C
`MOV.B`	`#20,R7`	`;copia o valor 20 decimal para R7 (R7 = 20).`	?	?	?	?
`SUB.B`	`#10,R7`	`;subtrai 10 decimal do conteúdo de R7 (R7 = 20 - 10 = 10). Observe que o flag C é setado para indicar que não houve necessidade de empréstimo (o resultado é positivo).`	0	0	0	1
`SUB.B`	`#15,R7`	`;subtrai 15 decimal do conteúdo de R7 (R7 = 10 - 15 = -5 ou 0x00FB em complemento de dois 8 bits). Repare que o flag C = 0 indica que houve necessidade de um empréstimo para se completar a subtração, o que indica que o resultado é negativo. O flag N = 1 também sinaliza esta situação.`	0	1	0	0

4.4.2.7. SUBC — Subtrai a fonte e o transporte do destino

Formato: SUBC fonte,destino ou SUBC.W fonte,destino

SUBC.B fonte,destino

Essa instrução subtrai o conteúdo do operando "fonte" e o *carry* do conteúdo do operando "destino".

A operação de subtração é realiza pela soma do conteúdo de "destino" e o complemento de dois do operando "fonte" mais o *carry* (C). Ou seja, "destino" = "destino" + NOT "fonte" + C, ou ainda: "destino" = "destino" – "fonte" – 1 + C.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Observação: O mnemônico SBB pode ser utilizado como sinônimo de SUBC.

Op-code: (instrução com 2 operandos): 4 *bits* - 0111.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o resultado da operação envolvendo operandos sinalizados ultrapasse os limites de representação (menor que –32.768 ou maior que +32.767 para operações de 16 *bits* e menor que –128 ou maior que +127 para operações de 8 *bits*);

N - setado se o resultado da operação for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso não tenha sido necessário um empréstimo, ou seja, resultado positivo. Apagado caso tenha sido necessário um empréstimo (resultado negativo).

Exemplos:			*Flags* alterados V	N	Z	C
MOV	#0x0001,R4	;R4 = 0x0001.	?	?	?	?
CLR	R5	;R5 = 0.	?	?	?	?
CLR	R6	;R6 = 0.	?	?	?	?
MOV	#0xFFFF,R7	;R7 = 0xFFFF.	?	?	?	?

As instruções seguintes subtraem o valor de 32 *bits* armazenado em R4 e R5 (0x00010000) do valor de 32 *bits* armazenado em R6 e R7 (0x0000FFFF):

SUB	R5,R7	;subtrai o conteúdo de R5 do conteúdo de R7. O resultado é armazenado em R7 (R7 = 0xFFFF). Repare que o flag C = 1, indi-cando que não houve empréstimo.	0	1	0	1
SUBC	R4,R6	;subtrai o conteúdo de R4 do conteúdo de R6 e soma o Carry (C). R6 = 0 - 1 - 1 + 1 = 0xFFFF. Repare que o flag C = 0, indicando que houve empréstimo (resultado negativo).	0	1	0	0

4.4.2.8. SBC

Subtrai o transporte do destino

Formato: SBC destino ou SBC.W destino

SBC.B destino

Essa instrução subtrai o conteúdo do *flag* de transporte (atuando como empréstimo) do conteúdo do operando "destino".

A operação de subtração é realizada pela soma do conteúdo de "destino" com o *carry* (C) mais a constante 0xFF ou 0xFFFF, conforme a operação (8 ou 16 *bits*). A adição da constante 0xFF ou 0xFFFF equivale à soma da constante –1.

Também podemos descrever a operação como "destino" = "destino" - (NOT C).

Na prática, a operação é emulada pela instrução SUBC #0,destino.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: emulada pela instrução SUBC.

***Flags* afetados no SR:** V, N, Z e C.

- V - setado caso o resultado da operação envolvendo operandos sinalizados ultrapasse os limites de representação (menor que –32.768 ou maior que +32.767 para operações de 16 *bits* e menor que –128 ou maior que +127 para operações de 8 *bits*);
- N - setado se o resultado da operação for negativo, apagado caso seja positivo;
- Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;
- C - setado caso não tenha sido necessário um empréstimo, ou seja, resultado positivo. Apagado caso tenha sido necessário um empréstimo (resultado negativo).

Exemplos:			*Flags* alterados V	N	Z	C
MOV	#0x3000,R4	;R4 = 0x3000.	?	?	?	?
BIC	#1,SR	;apaga o flag C (C = 0).	?	?	?	0
SBC	R4	;subtrai C de R4. A operação é: R4 = 0x3000 + 0xFFFF + 0 = 0x2FFF ou simplesmente R4 = 0x3000 - 1, onde 1 é o complemento do flag C.	0	0	0	1

4.4.2.9. INC — Incrementa o destino

Formato: INC destino ou INC.W destino
INC.B destino

A instrução INC é utilizada para incrementar em um o conteúdo do operando "destino", ou seja, "destino" = "destino" + 1.

Essa instrução não existe fisicamente, sendo emulada pela instrução ADD #1,destino.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: emulada pela instrução ADD.

***Flags* afetados no SR:** V, N, Z e C.

- V - setado caso o destino contivesse o valor 0x7FFF (operação de 16 *bits*) ou 0x??7F (operação de 8 *bits*) antes da execução da instrução;
- N - setado se o resultado da operação for negativo, apagado caso seja positivo;
- Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;
- C - setado caso a operação de incremento tenha provocado um transbordo do conteúdo do registrador (de 0xFFFF para 0x0000 ou de 0x00FF para 0x0000, conforme a operação seja de 16 ou de 8 *bits* respectivamente).

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV`	`#0x3000,R4`	`;R4 = 0x3000.`	?	?	?	?
`INC`	`R4`	`;incrementa R4 (R4 = 0x3001).`	0	0	0	0
`MOV`	`#0x30FF,R5`	`;R5 = 0x30FF. Os flags não são alterados e permanecem nos seus estados anteriores.`	0	0	0	0
`INC.B`	`R5`	`;incrementa R5 em uma operação de 8 bits. Neste caso, a operação ocorre apenas em relação ao byte menos significativo do registrador, o byte superior é zerado. R5 = 0x0000.`	0	0	1	1

4.4.2.10. INCD — Incrementa o destino em dois

Formato: INCD destino ou INCD.W destino
INCD.B destino

A instrução INCD (duplo incremento) é utilizada para incrementar em dois o conteúdo do operando "destino", ou seja, "destino" = "destino" + 2.

Essa instrução não existe fisicamente, sendo emulada pela instrução ADD #2,destino.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: emulada pela instrução ADD.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o destino contivesse o valor 0x7FFE (operação de 16 *bits*) ou 0x??7E (operação de 8 *bits*) antes da execução da instrução;

N - setado se o resultado da operação for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso a operação de incremento tenha provocado um transbordo do conteúdo do registrador (de 0xFFFF para 0x0000 ou de 0x00FF para 0x0000, conforme a operação seja de 16 ou de 8 *bits* respectivamente).

Exemplos:			***Flags* alterados** V	N	Z	C
MOV	#0x3000,R4	;R4 = 0x3000.	?	?	?	?
INC	R4	;incrementa R4 em dois (R4 = 0x3002).	0	0	0	0
MOV	#0x30FF,R5	;R5 = 0x30FF. Os flags não são alterados e permanecem nos seus estados anteriores.	0	0	0	0
INC.B	R5	;incrementa R5 em uma operação de 8 bits. Neste caso, a operação ocorre apenas em relação ao byte menos significativo do registrador, o byte superior é zerado. R5 = 0x0001.	0	0	0	1

4.4.2.11. DEC

Decrementa o destino

Formato: DEC destino ou DEC.W destino

DEC.B destino

A instrução DEC é utilizada para decrementar em um o conteúdo do operando "destino", ou seja, "destino" = "destino" – 1.

Essa instrução não existe fisicamente, sendo emulada pela instrução SUB #1,destino. Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: emulada pela instrução SUB.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o destino contivesse o valor 0x8000 (operação de 16 *bits*) ou 0x??80 (operação de 8 *bits*) antes da execução da instrução;

N - setado se o resultado da operação for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso o destino contivesse zero antes da operação.

Exemplos:			***Flags* alterados** V	N	Z	C
MOV	#0x3000,R4	;R4 = 0x3000.	?	?	?	?
DEC.B	R4	;decrementa R4 em uma operação de 8 bits (R4 = 0x00FF). Observe que N terminou setado porque o resultado de 8 bits da operação (0xFF) está com o seu bit 7 setado.	0	1	0	0

4.4.2.12. DECD

Decrementa o destino em dois

Formato: DECD destino ou DECD.W destino
DECD.B destino

A instrução DECD é utilizada para decrementar em dois o conteúdo do operando "destino", ou seja, "destino" = "destino" – 2.

Essa instrução não existe fisicamente, sendo emulada pela instrução SUB #2,destino.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: emulada pela instrução SUB.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o destino contivesse o valor 0x8.001 (operação de 16 *bits*) ou 0x??81 (operação de 8 *bits*) antes da execução da instrução;

N - setado se o resultado da operação for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso o destino contivesse zero antes da operação.

Exemplos:

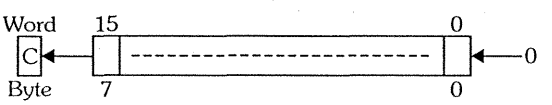
			***Flags* alterados**			
			V	**N**	**Z**	**C**
MOV	#0x3000,R4	;R4 = 0x3000.	?	?	?	?
DECD.B	R4	;decrementa R4 em dois, em uma operação de 8 bits (R4 = 0x00FE). Observe que N terminou setado porque o resultado de 8 bits da operação (0xFE) está com o seu bit 7 setado.	0	1	0	0

4.4.2.13. RLA

Rotação aritmética à esquerda

Formato: RLA destino ou RLA.W destino
RLA.B destino

Essa instrução realiza a rotação ou deslocamento aritmético à esquerda do conteúdo de "destino", o que equivale a multiplicar o conteúdo de "destino" por 2, ou seja, "destino" = "destino" * 2.



A operação é emulada pelo uso da instrução ADD destino,destino, ou seja, soma-se o conteúdo de "destino" com ele mesmo, o que resulta na sua multiplicação por dois.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Observação: Essa instrução não pode ser utilizada com o modo de endereçamento indireto com auto-incremento, uma vez que esse modo não está disponível para o operando de destino. Em vez disso, o programador deve utilizar a instrução ADD e o modo indexado em –2 (operações de 16 *bits*) ou –1 (operações de 8 *bits*) como operando de destino.

Op-code: emulada pela instrução ADD.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o resultado da operação que envolva operandos sinalizados ultrapasse os limites de representação (menor que –32.768 ou maior que +32.767 para operações de 16 *bits* e menor que –128 ou maior que +127 para operações de 8 *bits*);

N - setado se o resultado da operação que envolve operandos sinalizados for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso ocorra um transporte do *bit* 15 (operação de 16 *bits*) ou do *bit* 7 (operação de 8 *bits*), indicando que o resultado é maior que 65.535 (16 *bits*) ou 255 (8 *bits*).

Exemplos:			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV.B`	`#0x37,R4`	`;R4 = 0x0037.`	?	?	?	?

`RLA.B`	`R4`	`;multiplica R4 por 2 em uma operação de 8 bits.`
		`R4 antes = 0000 0000 0011 0111`
		`R4 depois = 0000 0000 0110 1110`
		`R4 = 0x006E.`
			0	0	0	0

`RLA.B`	`R4`	`;multiplica R4 por 2 em uma operação de 8 bits.`
		`R4 antes = 0000 0000 0110 1110`
		`R4 depois = 0000 0000 1101 1100`
		`R4 = 0x00DC.`
			0	1	0	0

`RLA.B`	`R4`	`;multiplica R4 por 2 em uma operação de 8 bits.`
		`R4 antes = 0000 0000 1101 1100`
		`R4 depois = 0000 0000 1011 1000`
		`R4 = 0x00B8. Repare que o flag C terminou setado pois ocorreu um transporte do bit 7 do valor de 8 bits em R4, o que indica que o resultado ultrapassou 255.`
			0	1	0	1

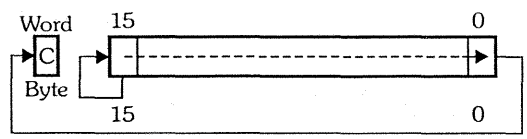
`RLA`	`R4`	`;multiplica R4 por 2 em uma operação de 16 bits.`
		`R4 antes = 0000 0000 1011 1000`
		`R4 depois = 0000 0001 0111 0000`
		`R4 = 0x0170.`
			0	0	0	0

4.4.2.14. RRA — Rotação aritmética à direita

Formato: RRA destino ou RRA.W destino
RRA.B destino

Essa instrução realiza a rotação ou deslocamento aritmético à direita do conteúdo de "destino", o que equivale à divisão inteira do conteúdo de "destino" por 2, ou seja, "destino" = "destino"/2.

O *bit* de sinal (*bit* 15) de "destino" é preservado e realimentado a cada operação:



Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com um operando): 8 *bits* - 00010001.

***Flags* afetados no SR:** V, N, Z e C.

V - sempre em zero;

N - setado se o resultado da operação que envolve operandos sinalizados for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - carregado com o conteúdo do *bit* 0 de "destino" antes da operação;

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV.B`	`#0x37,R4`	`;R4 = 0x0037.`	?	?	?	?

`RRA.B`	`R4`	`;divide R4 por 2 em uma operação de 8 bits.`
		`R4 antes = 0000 0000 0011 0111`
		`R4 depois = 0000 0000 0001 1011`
		`R4 = 0x001B.`
			0	0	0	1

`MOV`	`#0x0852,R4`	`;R4 = 0x0852;`	0	0	0	1

`RRA`	`R4`	`;divide R4 por 2 em uma operação de 16 bits.`
		`R4 antes = 0000 1000 0101 0010`
		`R4 depois = 0000 0100 0010 1001`
		`R4 = 0x0429.`
			0	0	0	0

4.4.2.15. SXT — Extensão do sinal

Formato: SXT destino

Essa instrução provoca a propagação do *bit* de sinal de um valor de 8 *bits* (ou seja, o estado do *bit* 7) para o *byte* mais significativo do destino de 16 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com 1 operando): 10 *bits* - 0001000110.

***Flags* afetados no SR:** V, N, Z e C.

V - sempre em zero;

N - setado se o resultado da operação for negativo (MSB setado), apagado caso seja positivo (MSB apagado);

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso o resultado da operação (8 ou 16 *bits*) seja diferente de zero, apagado caso seja igual a zero (estado contrário do *flag* Z).

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV.B`	`#0xFE,R4`	`;R4 = 0x00FE (equivalente a -2 em 8 bits).`	?	?	?	?
`SXT`	`R4`	`;propaga o sinal do bit 7 (igual a 1) para o byte mais significativo. R4 = 0xFFFE (equivalente a -2 em 16 bits).`	0	1	0	1
`MOV.B`	`#0x7F,R4`	`;R4 = 0x007F (127 positivo).`	0	0	0	0
`SXT`	`R4`	`;propaga o sinal do bit 7 (igual a 0) para o byte mais significativo. R4 = 0x007F (127 positivo).`	0	0	0	1

4.4.2.16. AND

Operação lógica "E"

Formato: AND fonte,destino ou AND.W fonte,destino

AND.B fonte,destino

Essa instrução realiza a operação lógica "E" ou AND *bit* a bit, entre o operando "fonte" e o operando "destino", armazenando o resultado em "destino".

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com 2 operandos): 4 *bits* - 1111.

***Flags* afetados no SR:** V, N, Z e C.

V - sempre apagado ("0");

N - setado se o resultado da operação for negativo (MSB setado), apagado caso seja positivo (MSB apagado);

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso o resultado da operação (8 ou 16 *bits*) seja diferente de zero, apagado caso seja igual a zero (estado contrário do *flag* Z).

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV`	`#0x1234,R4`	`;R4 = 0x1234.`	?	?	?	?
`AND`	`#0x0F0F,R4`	`;realiza a operação lógica "E" entre os dezesseis bits de R4 e os dezesseis bits do operando imediato 0x0F0F, o resultado é armazenado em R4 (R4 = 0x0204).`	0	0	0	1

4.4.2.17. XOR

Operação lógica "EOU"

Formato: XOR fonte,destino ou XOR.W fonte,destino

XOR.B fonte,destino

Essa instrução realiza a operação lógica "OU exclusivo" ou XOR *bit* a bit, entre o operando "fonte" e o operando "destino", armazenando o resultado em "destino".

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com 2 operandos): 4 *bits* - 1110.

***Flags* afetados no SR:** V, N, Z e C.

V - setado se ambos os operandos forem negativos (MSB setado em ambos);

N - setado se o resultado da operação for negativo (MSB setado), apagado caso seja positivo (MSB apagado);

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso o resultado da operação (8 ou 16 *bits*) seja diferente de zero, apagado caso seja igual a zero (estado contrário do *flag* Z).

Exemplos:			*Flags* alterados V	N	Z	C
MOV	#0x1234,R4	;R4 = 0x1234.	?	?	?	?
XOR	#0xF00F,R4	;realiza a operação lógica "EOU" entre os dezesseis bits de R4 e os dezesseis bits do operando imediato 0xF00F, o resultado é armazenado em R4 (R4 = 0xE23B). Na prática, para cada posição com um bit setado no operando fonte, o bit da respectiva posição no operando destino é invertido.	0	1	0	1

4.4.2.18. INV

Complementa os *bits* do destino

Formato: INV destino ou INV.W destino
INV.B destino

Essa instrução realiza a operação lógica "NÃO" ou NOT do conteúdo do operando "destino", ou seja, os *bits* de "destino" têm o seu estado invertido.

A instrução INV não possui um *op-code* específico, mas é emulada pela instrução XOR #0xFFFF,destino ou XOR.B #0xFF,destino.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: emulada pela instrução XOR.

***Flags* afetados no SR:** V, N, Z e C.

- V - setado se ambos os operandos forem negativos (MSB setado em ambos);
- N - setado se o resultado da operação for negativo (MSB setado), apagado caso seja positivo (MSB apagado);
- Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;
- C - setado caso o resultado da operação (8 ou 16 *bits*) seja diferente de zero, apagado caso seja igual a zero (estado contrário do *flag* Z).

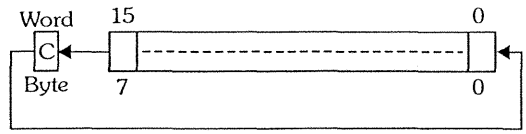
Exemplos:			*Flags* alterados V	N	Z	C
MOV	#0x1234,R4	;R4 = 0x1234.	?	?	?	?
INV	R4	;inverte os bits de R4. Após a operação, R4 = 0xEDCB.	0	1	0	1

4.4.2.19. RLC

Rotação lógica à esquerda pelo *carry*

Formato: RLC destino ou RLC.W destino
RLC.B destino

Essa instrução realiza a rotação ou deslocamento lógico à esquerda do conteúdo de "destino" pelo *carry*, ou seja, os *bits* de "destino" são deslocados uma posição à esquerda, sendo o *bit* 0 posteriormente preenchido com o conteúdo do *carry* (C), enquanto o *bit* excedente (o antigo MSB) é armazenado no *carry*:



A operação é emulada com o uso da instrução ADDC destino,destino, ou seja, soma-se o conteúdo de "destino" com ele mesmo, o que resulta na sua multiplicação por dois.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Observação: Essa instrução não pode ser utilizada com o modo de endereçamento indireto com auto-incremento, uma vez que esse modo não está disponível para o operando de destino. Em vez disso, o programador deve utilizar a instrução ADDC e o modo indexado em –2 (operações de 16 *bits*) ou –1 (operações de 8 *bits*) como operando de destino.

Op-code: emulada pela instrução ADDC.

***Flags* afetados no SR:** V, N, Z e C.

V - setado caso o resultado da operação que envolve operandos sinalizados ultrapasse os limites de representação (menor que –32.768 ou maior que +32.767 para operações de 16 *bits* e menor que –128 ou maior que +127 para operações de 8 *bits*);

N - setado se o resultado da operação que envolve operandos sinalizados for negativo, apagado caso seja positivo;

Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;

C - setado caso ocorra um transporte do *bit* 15 (operação de 16 *bits*) ou do *bit* 7 (operação de 8 *bits*), indicando que o resultado é maior que 65.535 (16 *bits*) ou 255 (8 *bits*).

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV.B`	`#0x37,R4`	`;R4 = 0x0037.`	?	?	?	?
`BIS`	`#1,SR`	`;seta o carry (C=1).`	?	?	?	1

`RLC.B`	`R4`	`;rotaciona R4 à esquerda (8 bits):`
		`R4 antes = 0000 0000 0011 0111`
		`R4 depois = 0000 0000 0110 1111`
		`R4 = 0x006F. Repare que o flag C antes da operação era igual a 1 e este valor foi colocado na posição do bit 0 após o deslocamento. O bit 7 excedente da rotação (o antigo MSB) que era igual a zero, foi armazenado em C (C=0 após a operação).`
			0	0	0	0

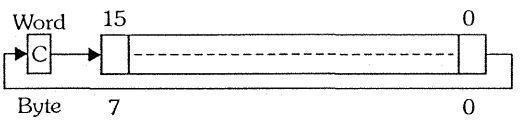
`MOV`	`#0xABC0,R5`	`;R5 = 0xABC0.`	0	0	0	0

`RLC`	`R5`	`;rotaciona R5 à esquerda (16 bits):`
		`R5 antes = 1010 1011 1100 0000`
		`R5 depois = 0101 0111 1000 0000`
		`R4 = 0x5780. Repare que o flag C antes da operação era igual a 0 e este valor foi colocado na posição do bit 0 após o deslocamento. O bit 15 excedente da rotação (o antigo MSB) que era igual a um, foi armazenado em C (C=1 após a operação).`
			0	0	0	1

4.4.2.20. RRC — Rotação lógica à direita

Formato: RRC destino ou RRC.W destino
RRC.B destino

Essa instrução realiza a rotação ou deslocamento lógico do conteúdo de "destino", um *bit* à direita por meio do *carry* (C), ou seja, o conteúdo de 8 ou 16 *bits* de "destino" é deslocado um *bit* à direita, o conteúdo do *flag* C é armazenado no *bit* mais significativo (*bit* 15 ou *bit* 7, conforme a operação seja de 16 ou 8 *bits*). O *bit* excedente da rotação (o *bit* 0 de "destino" antes da operação) é armazenado no *flag* C:



Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com um operando): 8 *bits* - 00010000.

***Flags* afetados no SR:** V, N, Z e C.

- V - setado se inicialmente o valor de "destino" for positivo e o *flag* C = 1;
- N - setado se o resultado da operação que envolve operandos sinalizados for negativo, apagado caso seja positivo;
- Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;
- C - carregado com o conteúdo do *bit* 0 de "destino" antes da operação.

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV.B`	`#0x37,R4`	`;R4 = 0x0037.`	?	?	?	?
`BIS`	`#1,SR`	`;seta o carry (C=1).`	?	?	?	1

`RRC.B`	`R4`	`;rotaciona R4 à direita (8 bits):`
		`R4 antes = 0000 0000 0011 0111`
		`R4 depois = 0000 0000 1001 1011`
		`R4 = 0x009B. Repare que o flag C antes da operação era igual a 1 e este valor foi colocado na posição do bit 7 após o deslocamento. O bit 0 excedente da rotação (o antigo LSB) que era igual a um, foi armazenado em C (C=1 após a operação).`
			0	1	0	1

`MOV`	`#0xABC0,R5`	`;R5 = 0xABC0.`	0	1	0	1

`RRC`	`R5`	`;rotaciona R5 à direita (16 bits):`
		`R5 antes = 1010 1011 1100 0000`
		`R5 depois = 1101 0101 1110 0000`
		`R4 = 0xD5E0. Repare que o flag C antes da operação era igual a 1 e este valor foi colocado na posição do bit 15 após o deslocamento. O bit 0 excedente da rotação (o antigo LSB) que era igual a zero, foi armazenado em C (C=0 após a operação).`
			0	1	0	0

4.4.3. Instruções de Teste e Desvio

Neste tópico vamos estudar as instruções que realizam testes condicionais e desvios do fluxo do programa.

A CPU dos MSP430 pode realizar os seguintes testes:

- Teste de *bit*;
- Teste de registrador (comparação com zero);
- Comparação de dois valores (menor, igual ou maior).

Com relação aos desvios, a CPU suporta as seguintes classes de desvio:

- Desvio para endereço absoluto;
- Desvio para endereço relativo;
- Desvio condicional para endereço relativo;
- Chamada de sub-rotina;
- Retorno de sub-rotina ou interrupção.

4.4.3.1. BIT — Testa um ou mais *bits*

Formato: BIT fonte,destino ou BIT.W fonte,destino
BIT.B fonte,destino

Essa instrução realiza uma operação lógica E (AND) entre os operandos "fonte" e "destino". Os *flags* são alterados de acordo com o resultado da operação, que é descartado. O conteúdo de "fonte" e "destino" não é alterado.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com 2 operandos): 4 *bits* - 1011.

***Flags* afetados no SR:** V, N, Z e C.

V - sempre apagado;
N - setado se o resultado da operação que envolve operandos sinalizados for negativo, apagado caso seja positivo;
Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero, apagado caso seja diferente de zero;
C - setado caso um dos *bits* esteja setado, apagado no caso contrário.

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV`	`#0xABCD,R6`	`;R6 = 0xABCD ou 1010 1011 1100 1101.`	?	?	?	?
`BIT`	`#1,R6`	`;testa o bit 0 de R6. Repare que como o bit 0 de R6 está setado, o resultado em C será igual a 1.`	0	0	0	1
`BIT`	`#0x1000,R6`	`;testa se o bit 12 de R6 está setado. Neste caso, o resultado em C será 0, pois o bit 12 está apagado. Z = 1, pois o resultado da operação AND é igual a zero.`	0	0	1	0

4.4.3.2. CMP — Compara os conteúdo dos operandos

Formato: CMP fonte,destino ou CMP.W fonte,destino

CMP.B fonte,destino

Subtrai o conteúdo do operando "fonte" do conteúdo do operando "destino". Os *flags* são alterados de acordo com o resultado da operação, que é descartado. O conteúdo de "fonte" e de "destino" não é alterado.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: (instrução com 2 operandos): 4 *bits* - 1001.

***Flags* afetados no SR:** V, N, Z e C.

- V - setado caso o resultado da operação que envolve operandos sinalizados ultrapasse os limites de representação (menor que –32.768 ou maior que +32.767 para operações de 16 *bits* e menor que –128 ou maior que +127 para operações de 8 *bits*);
- N - setado se o resultado da operação for negativo ("destino" < "fonte"), apagado caso seja positivo ("destino" >= "fonte");
- Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero ("fonte" = "destino"), apagado caso seja diferente de zero ("fonte" <> "destino");
- C - setado se "destino" >= "fonte".

Exemplos:			***Flags* alterados** V	N	Z	C
`MOV.B`	`#0x37,R4`	`;R4 = 0x37.`	?	?	?	?
`CMP.B`	`#0x36,R4`	`;compara o valor de R4 com o valor 0x36. Como R4 é maior que 0x36, C = 1.`	0	0	0	1
`CMP.B`	`#0x37,R4`	`;compara o valor de R4 com o valor 0x37. Como R4 é igual a 0x37, C = 1 e Z = 1.`	0	0	1	1
`CMP.B`	`#0x38,R4`	`;compara o valor de R4 com o valor 0x38. Como R4 é menor que 0x38, C = 0, Z = 0 e N = 1.`	0	1	0	0

4.4.3.3. TST — Testa se for igual a zero

Formato: TST destino ou TST.W destino

TST.B destino

Compara o conteúdo de "destino" com zero.

Essa instrução não possui um *op-code* próprio sendo emulada pela instrução CMP #0,destino.

Quando utilizada sem sufixo ou com o sufixo .W, a instrução realiza uma operação de 16 *bits*, ao passo que quando utilizada com o sufixo .B, a operação é de 8 *bits*.

Os *flags* matemáticos são alterados de acordo com o resultado da operação, conforme descrito em seguida.

Op-code: emulada pela instrução CMP.

***Flags* afetados no SR:** V, N, Z e C.

- V - sempre apagado;
- N - setado se o resultado da operação for negativo ("destino" < 0), apagado caso seja positivo ("destino" >= 0);
- Z - setado caso o resultado da operação (8 ou 16 *bits*) seja igual a zero ("fonte" = "destino"), apagado caso seja diferente de zero ("fonte" <> "destino");
- C - sempre setado.

Exemplos:			*Flags* alterados V	N	Z	C
MOV.B	#0x55,R4	;R4 = 0x55.	?	?	?	?
TST.B	R4	;testa se R4 = 0. O flag Z = 0 indica que o conteúdo de R4 é diferente de zero.	0	0	0	1

4.4.3.4. BR — Desvio absoluto

Formato: BR destino

Desvia o programa para um endereço especificado por "destino", que pode ser qualquer endereço da memória.

Sua operação é emulada pela instrução MOV destino,PC. Todos os modos de endereçamento válidos para operandos fonte podem ser utilizados com essa instrução.

Não há opção de 8 *bits* para essa instrução, pois todas as operações são de 16 *bits*.

Op-code: emulada pela instrução MOV.

***Flags* afetados no SR:** nenhum.

Exemplos:			*Flags* alterados V	N	Z	C
MOV.B	#0x3000,R4	;R4 = 0x3000.	?	?	?	?
BR	R4	;desvia para o endereço indicado por R4 (endereço 0x3000).	?	?	?	?
BR	#0x1000	;desvia para o endereço 0x1000.	?	?	?	?
BR	&0x2000	;desvia para o endereço de 16 bits contido em 0x2000 (byte menos significativo) e 0x2001 (byte mais significativo).	?	?	?	?
BR	TESTE	;desvia para o endereço do rótulo TESTE	?	?	?	?

4.4.3.5. JMP — Desvio incondicional

Formato: JMP destino

Desvia o programa para um endereço especificado em "destino".

O endereço de destino pode estar no máximo a 512 *words* à frente ou 511 *words* atrás da instrução de desvio.

Op-code: (instrução de desvio): 6 *bits* - 001111.

***Flags* afetados no SR:** nenhum.

Exemplo:			*Flags* alterados V	N	Z	C
JMP	TESTE	;desvia para o endereço apontado pelo rótulo TESTE.	?	?	?	?

4.4.3.6. JEQ/JZ — Desvia se igual

Formato: JEQ destino ou JZ destino

Desvia o programa para um endereço especificado em "destino" se o *flag* Z = 1. Após uma operação de subtração ou comparação, Z = 1 indica que o "destino" é igual ao "fonte".

O endereço de destino pode estar no máximo a 512 *words* à frente ou 511 *words* atrás da instrução de desvio.

Op-code: (instrução de desvio): 6 *bits* - 001001.

***Flags* afetados no SR:** nenhum.

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV.B`	`#0x37,R4`	`;R4 = 0x37.`	?	?	?	?
`CMP.B`	`#0x37,R4`	`;compara o valor de R4 com o valor 0x37. Como R4 é igual a 0x37, C = 1 e Z = 1.`	0	0	1	1
`JEQ`	`#0xF900`	`;desvia para o endereço 0xF900, pois Z = 1.`	0	0	1	1

4.4.3.7. JNE/JNZ — Desvia se diferente

Formato: JNE destino ou JNZ destino

Desvia o programa para um endereço especificado em "destino" se o *flag* Z = 0. Após uma operação de subtração ou comparação, Z = 0 indica que o "destino" é diferente de "fonte".

O endereço de destino pode estar no máximo a 512 *words* à frente ou 511 *words* atrás da instrução de desvio.

Op-code: (instrução de desvio): 6 *bits* - 001000.

***Flags* afetados no SR:** nenhum.

Exemplos:

			***Flags* alterados**			
			V	**N**	**Z**	**C**
`MOV.B`	`#0x37,R4`	`;R4 = 0x37.`	?	?	?	?
`CMP.B`	`#0x37,R4`	`;compara o valor de R4 com o valor 0x37. Como R4 é igual a 0x37, C = 1 e Z = 1.`	0	0	1	1
`JNE`	`TESTE`	`; não desvia para TESTE, pois Z = 1.`	0	0	1	1

4.4.3.8. JC/JHS — Desvia se maior ou igual (sem sinal)

Formato: JC destino ou JHS destino

Desvia o programa para um endereço especificado em "destino" se o *flag* C = 1. Após uma operação de subtração ou comparação com números não sinalizados, C = 1 indica que o "destino" é maior ou igual ao "fonte".

O endereço de destino pode estar no máximo a 512 *words* à frente ou 511 *words* atrás da instrução de desvio.

Op-code: (instrução de desvio): 6 *bits* - 001011.

***Flags* afetados no SR:** nenhum.

Exemplos:			*Flags* alterados V	N	Z	C
MOV	#0xABCD,R6	;R6 = 0xABCD ou 1010 1011 1100 1101.	?	?	?	?
BIT	#1,R6	;testa o bit 0 de R6. Repare que como o bit 0 de R6 está setado, o resultado em C será igual a 1.	0	0	0	1
JC	TESTE	;desvia para TESTE, pois C = 1.	0	0	0	1

4.4.3.9. JNC/JLO

Desvia se for menor (sem sinal)

Formato: JNC destino ou JLO destino

Desvia o programa para um endereço especificado em "destino" se o *flag* C = 0. Após uma operação de subtração ou comparação com números não sinalizados, C = 0 indica que o "destino" é menor que "fonte".

O endereço de destino pode estar no máximo a 512 *words* à frente ou 511 *words* atrás da instrução de desvio.

Op-code: (instrução de desvio): 6 *bits* - 001010.

***Flags* afetados no SR:** nenhum.

Exemplos:			*Flags* alterados V	N	Z	C
MOV	#0xABCD,R6	;R6 = 0xABCD ou 1010 1011 1100 1101.	?	?	?	?
BIT	#1,R6	;testa o bit 0 de R6. Repare que como o bit 0 de R6 está setado, o resultado em C será igual a 1.	0	0	0	1
JNC	TESTE	;não desvia para TESTE, pois C = 1.	0	0	0	1

4.4.3.10. JGE

Desvia se for maior ou igual (sinalizado)

Formato: JGE destino

Desvia o programa para um endereço especificado em "destino" se os *flags* N = 1 e V = 1 ou se N = 0 e V = 0. Se após uma operação de subtração ou comparação com números sinalizados o *flag* N = V, significa que o valor do operando "destino" é maior ou igual ao do operando "fonte".

O endereço de destino pode estar no máximo a 512 *words* à frente ou 511 *words* atrás da instrução de desvio.

Op-code: (instrução de desvio): 6 *bits* - 001101.

***Flags* afetados no SR:** nenhum.

Exemplos:			*Flags* alterados V	N	Z	C
MOV.B	#-2,R4	;R4 = 0x00FE (-2 em 8 bits).	?	?	?	?
MOV.B	#5,R5	;R5 = 0x0005.	?	?	?	?
CMP.B	R4,R5	;compara R4 com R5 (R5 - R4). Como 5 > -2, V = N e Z e C = 0.	0	0	0	0
JGE	SUB1	;desvia para o endereço do rótulo SUB1, pois a última comparação fez com que N = V.	0	0	0	0

4.4.3.11. JL — Desvia se for menor (sinalizado)

Formato: JL destino

Desvia o programa para um endereço especificado em "destino" se os *flags* N = 1 e V = 0 ou se N = 0 e V = 1. Se após uma operação de subtração ou comparação com números sinalizados o *flag* N <> V, significa que o valor do operando "destino" é menor que o do operando "fonte".

O endereço de destino pode estar no máximo a 512 *words* à frente ou 511 *words* atrás da instrução de desvio.

Op-code: (instrução de desvio): 6 *bits* - 001110.

***Flags* afetados no SR:** nenhum.

Exemplos:			***Flags* alterados** V	N	Z	C
`MOV.B`	`#-2,R4`	`;R4 = 0x00FE (-2 em 8 bits).`	?	?	?	?
`MOV.B`	`#5,R5`	`;R5 = 0x0005.`	?	?	?	?
`CMP.B`	`R5,R4`	`;compara R5 com R4 (R4 - R5). Como R4 < R5 (-2 < 5), N <> V.`	0	1	0	1
`JL`	`TESTE`	`;desvia para o endereço apontado pelo rótulo TESTE, pois a última comparação resultou N <> V.`	0	1	0	1

4.4.3.12. JN — Desvia se for negativo

Formato: JN destino

Desvia o programa para um endereço especificado em "destino" se o *flag* N = 1.

O endereço de destino pode estar no máximo a 512 *words* à frente ou 511 *words* atrás da instrução de desvio.

Op-code: (instrução de desvio): 6 *bits* - 001100.

***Flags* afetados no SR:** nenhum.

Exemplos:			***Flags* alterados** V	N	Z	C
`MOV.B`	`#-2,R4`	`;R4 = 0x00FE (-2 em 8 bits).`	?	?	?	?
`MOV.B`	`#5,R5`	`;R5 = 0x0005.`	?	?	?	?
`CMP.B`	`R5,R4`	`;compara R5 com R4 (R4 - R5). Como o resultado é negativo, N = 1.`	0	1	0	1
`JN`	`TESTE`	`;desvia para o endereço apontado pelo rótulo TESTE, pois a última comparação resultou negativo (N=1).`	0	1	0	1

4.4.3.13. CALL — Chamada de sub-rotina

Formato: CALL destino

Realiza a chamada de uma sub-rotina localizada no endereço especificado por "destino", que pode ser qualquer endereço da memória.

As operações realizadas por essa instrução são (em seqüência):

- O endereço de destino é lido;
- O SP é decrementado em dois;
- O PC atual, que aponta para a instrução seguinte ao CALL, é armazenado na pilha (este é o endereço de retorno da sub-rotina);
- O endereço da sub-rotina é armazenado no PC.

Todos os modos de endereçamento válidos para operandos fonte podem ser utilizados com essa instrução.

Op-code: (instrução com 1 operando): 9 *bits* - 000100101.

***Flags* afetados no SR:** nenhum.

Exemplos:

			Flags alterados V	N	Z	C
`CALL`	`SUB1`	`;chamada da sub-rotina SUB1`	?	?	?	?
`CALL`	`R9`	`;chama a sub-rotina localizada no endereço contido em R9.`	?	?	?	?
`CALL`	`@R9`	`;chama a sub-rotina apontada pelo endereço armazenado em R9.`	?	?	?	?

4.4.3.14. RET

Retorno de sub-rotina

Formato: RET

Provoca o retorno de uma sub-rotina.
O endereço de retorno armazenado no topo da pilha é lido e armazenado no PC.
Essa operação é emulada pela instrução MOV @SP+,PC.

Op-code: emulada pela instrução MOV.

***Flags* afetados no SR:** nenhum.

Exemplo:

		Flags alterados V	N	Z	C
`RET`	`;retorna da sub-rotina atual`	?	?	?	?

4.4.3.15. RETI

Retorno de uma interrupção

Formato: RETI

Provoca o retorno de uma rotina de tratamento de interrupção (RTI).
As operações realizas por essa instrução são (em seqüência):

- O valor do topo da pilha é lido e armazenado em SR (SR é restaurado);
- O SP é incrementado em dois;
- O conteúdo do topo da pilha é armazenado no PC (endereço de retorno);
- O SP é incrementado em dois.

Op-code: (instrução com um operando): 9 *bits* – 000100110.

***Flags* afetados no SR:** os *flags* retornam ao seu estado anterior à interrupção.

Exemplos:

		Flags alterados V	N	Z	C
`RETI`	`;retorna da interrupção.`	?	?	?	?

4.4.4. Instruções de Controle do Processador

As instruções de controle do processador destinam-se basicamente a controlar alguns aspectos do funcionamento da CPU.

4.4.4.1. CLRC



Apaga o *flag* C

Formato: CLRC

Apaga o *flag* C (C=0) no registrador SR.

Essa operação é emulada pela instrução BIC #1,SR.

Op-code: emulada pela instrução BIC.

***Flags* afetados no SR:** C.

C apagado;

Exemplos:

```
CLRC      ;apaga o flag C (C=0).
```

Flags alterados			
V	**N**	**Z**	**C**
?	?	?	0

4.4.4.2. CLRN



Apaga o *flag* N

Formato: CLRN

Apaga o *flag* N (N=0) no registrador SR.

Essa operação é emulada pela instrução BIC #4,SR.

Op-code: emulada pela instrução BIC.

***Flags* afetados no SR:** N.

N apagado;

Exemplos:

```
CLRN      ;apaga o flag N (N=0).
```

Flags alterados			
V	**N**	**Z**	**C**
?	0	?	?

4.4.4.3. CLRZ



Apaga o *flag* Z

Formato: CLRZ

Apaga o *flag* Z (Z=0) no registrador SR.

Essa operação é emulada pela instrução BIC #2,SR.

Op-code: emulada pela instrução BIC.

***Flags* afetados no SR:** Z.

Z apagado;

Exemplos:

```
CLRZ      ;apaga o flag Z (Z=0).
```

Flags alterados			
V	**N**	**Z**	**C**
?	?	0	?

4.4.4.4. SETC

Seta o *flag* C

Formato: SETC

Seta o *flag* C (C=1) no registrador SR.

Essa operação é emulada pela instrução BIS #1,SR.

Op-code: emulada pela instrução BIS.

***Flags* afetados no SR:** C.

C setado;

Exemplos:

```
SETC      ;seta o flag C (C=1).
```

Flags alterados			
V	**N**	**Z**	**C**
?	?	?	1

4.4.4.5. SETN

Seta o *flag* N

Formato: SETN

Seta o *flag* N (N=1) no registrador SR.

Essa operação é emulada pela instrução BIS #4,SR.

Op-code: emulada pela instrução BIS.

***Flags* afetados no SR:** N.

N setado;

Exemplos:

```
SETN      ;seta o flag N (N=1).
```

Flags alterados			
V	**N**	**Z**	**C**
?	1	?	?

4.4.4.6. SETZ

Seta o *flag* Z

Formato: SETZ

Seta o *flag* Z (Z=1) no registrador SR.

Essa operação é emulada pela instrução BIS #2,SR.

Op-code: emulada pela instrução BIS.

***Flags* afetados no SR:** Z.

Z setado;

Exemplos:

```
SETZ      ;seta o flag Z (Z=1).
```

Flags alterados			
V	**N**	**Z**	**C**
?	?	1	?

4.4.4.7. DINT  Desabilita interrupções

Formato: DINT

Apaga o *bit* GIE no SR (GIE = 0), desabilitando as interrupções mascaráveis.

Essa operação é emulada pela instrução BIC #8,SR.

Op-code: emulada pela instrução BIC.

***Flags* afetados no SR:** nenhum.

Exemplos:

		Flags alterados			
		V	N	Z	C
`DINT`	`;desabilita interrupções (GIE = 0).`	?	?	?	?

4.4.4.8. EINT  Habilita interrupções

Formato: EINT

Seta o *bit* GIE no SR (GIE = 1), habilitando as interrupções mascaráveis.

Essa operação é emulada pela instrução BIS #8,SR.

Op-code: emulada pela instrução BIS.

***Flags* afetados no SR:** nenhum.

Exemplos:

		Flags alterados			
		V	N	Z	C
`EINT`	`;habilita interrupções (GIE = 1).`	?	?	?	?

4.4.4.9. NOP  Nenhuma operação

Formato: NOP

A instrução NOP pode ser utilizada para gastar tempo da CPU em sub-rotinas de atraso ou controle de tempo, ou ainda para substituir instruções em testes de programa.

Essa instrução não está implementada fisicamente e é emulada pela instrução MOV #0,R3.

Além dela, existem diversas outras alternativas para esse tipo de operação, com tempos e tamanhos diferentes que serão vistos nos exemplos seguintes.

Op-code: emulada pela instrução MOV.

***Flags* afetados no SR:** nenhum.

Exemplos:

			Flags alterados			
			V	N	Z	C
`NOP`		`;nenhuma operação. Ocupa 1 word de memória e leva 1 ciclo para ser executada.`	?	?	?	?
`JMP`	`$+2`	`;desvia para a próxima instrução. Ocupa o mesmo espaço que um NOP, mas gasta 2 ciclos para ser executada.`	?	?	?	?
`MOV`	`&10,&10`	`;copia o conteúdo do endereço 10 para ele mesmo. Ocupa 3 words e gasta 6 ciclos para ser executada.`	?	?	?	?

4.5. Temporização das Instruções

Para encerrar o estudo das instruções *Assembly* dos MSP430, vamos analisar a temporização de execução das instruções vistas até aqui.

Como ponto de partida é importante sabermos que o período de execução de uma instrução é medido em ciclos do *clock* principal do sistema (MCLK) e que uma instrução pode levar de um a seis ciclos de *clock* para ser executada. O tempo exato varia de acordo com alguns fatores, como os seguintes:

1. O número e tipo de operandos presentes;
2. O modo de endereçamento utilizado.

Relativamente ao primeiro item, podemos classificar as instruções em quatro grupos:

1. Instruções sem operando (ocupam somente uma *word* de memória);
2. Instruções de desvio do tipo Jxx (ocupam somente uma *word* de memória);
3. Outras instruções com um operando (ocupam de uma a duas *words* de memória);
4. Outras instruções com dois operandos (ocupam de uma a três *words* de memória).

As instruções sem operando possuem tempo de execução fixo, conforme a tabela 4-21.

Instrução	Ciclos
CLRC, CLRN, CLRZ	1
SETC, SETN, SETZ	1
DINT, EINT	1
NOP	1
RET	3
RETI	5

Tabela 4-21

As instruções de desvio do tipo JUMP (JMP, JZ, JNZ, JC, JNC, JGE, JL e JN) necessitam sempre de dois ciclos de *clock* para a sua execução, independente de ocorrer ou não o desvio.

No caso das instruções com apenas um operando, utiliza-se a tabela 4-22 para o cálculo de tempo de execução. A tabela 4-23 demonstra a temporização no caso das instruções emuladas e com apenas um operando.

Instrução	Modo de endereçamento						
	Registrador Rn	Indireto @Rn	Indireto auto--incremento @Rn+	Imediato #n	Indexado x(Rn)	Sim-bólico ROT	Absoluto &ROT
RRA, RRC	1	3	3	-	4	4	4
SWPB, SXT	1	3	3	-	4	4	4
PUSH	3	4	4	4	5	5	5
CALL	4	4	5	5	5	5	5

Tabela 4-22

Instrução	Modo de endereçamento							
	Reg. Rn	Reg. PC	Indireto @Rn	Indireto auto--incremento @Rn+	Imediato #n	Indexado x(Rn)	Simbólico ROT	Absoluto &ROT
ADC	1	2	-	-	-	4	4	4
CLR	1	2	-	-	-	4	4	4
DADC	1	2	-	-	-	4	4	4
DEC	1	2	-	-	-	4	4	4
DECD	1	2	-	-	-	4	4	4
INC	1	2	-	-	-	4	4	4
INCD	1	2	-	-	-	4	4	4
INV	1	2	-	-	-	4	4	4
POP	1	2	-	-	-	4	4	4
SBC	1	2	-	-	-	4	4	4
TST	1	2	-	-	-	4	4	4
BR	2	2	3	3	3	3	3	3
RLA	1	2	-	-	-	6	6	6
RLC	1	2	-	-	-	6	6	6

Tabela 4-23

Com relação às demais instruções, o cálculo do tempo de execução pode ser feito com base nos modos de endereçamento utilizados, conforme a tabela 4-24:

Modo do operando fonte	Modo do operando destino				
	Rn	PC	x(Rn)	ROT	&ROT
Rn	1	2	4	4	4
@Rn	2	3	5	5	5
@Rn+	2	3	5	5	5
#n	2	3	5	5	5
x(Rn)	3	3	6	6	6
ROT	3	3	6	6	6
&ROT	3	3	6	6	6

Tabela 4-24

No caso da utilização do operando fonte no modo imediato (#n), o número de ciclos necessários será diminuído de um se for utilizada uma das constantes disponíveis nos geradores de constantes. Alguns exemplos de instruções e seus tempos de execução:

```
MOV     R3,R5         ; 1 ciclo
BIS     #1,R5         ; 1 ciclo (constante do CG2)
BIS     #16,R6        ; 2 ciclos (não utilizou o gerador de constantes)
MOV     #10,R9        ; 2 ciclos (não utilizou o gerador de constantes)
ADD     #20,R10       ; 2 ciclos (não utilizou o gerador de constantes)
RRC     @R9+          ; 3 ciclos
SWPB    TESTE         ; 4 ciclos
MOV     #1,TESTE      ; 4 ciclos (constante do CG2)
SUB     #10,TESTE     ; 5 ciclos
SBC     TESTE,AUX     ; 6 ciclos
```

5

Periféricos e Módulos Internos

Este capítulo apresenta os diversos periféricos encontrados nas famílias 1xx, 2xx e 4xx.

Nem todos os periféricos abordados estão disponíveis em um mesmo modelo de microcontrolador. A disponibilidade de cada um deles é apresentada no decorrer de cada tópico.

5.1. Sistema de *Reset*

O sistema de controle de *reset* e o sistema de interrupção confundem-se por utilizarem uma mesma estrutura de *hardware* básico. De fato, vamos verificar que o *reset* é tratado como uma interrupção não-mascarável de mais alta prioridade.

A arquitetura MSP430 possui dois sinais básicos relacionados ao *reset* do sistema:

- **POR** (***Power On Reset*** - *Reset* na energização do *chip*) - responsável pela inicialização de alguns registradores e condições do microcontrolador. O sinal POR é gerado em três condições:
  1. Na energização inicial do *chip*: o circuito do POR, ao detectar que a tensão de alimentação ultrapassou o limite mínimo necessário ao funcionamento do sistema (parâmetro $V_{MÍN}$ na folha de dados do *chip*, em média 0,2 Volts), o circuito POR gera um sinal que manterá o sistema em *reset* até que a tensão de alimentação ultrapasse o valor mínimo para funcionamento (parâmetro V_{POR} na folha de dados do *chip*). Esse valor é em média de 1,3 Volts. Após a tensão de alimentação ultrapassar o V_{POR}, o circuito de *reset* manterá o sinal de *reset* por aproximadamente 150μs e após esse período, libera o *chip* para funcionamento normal. Observe que o circuito POR somente volta a *ressetar* o sistema caso a tensão de alimentação caia abaixo de $V_{MÍN}$.
  2. Um nível lógico "0" aplicado ao pino $\overline{\text{RST/NMI}}$, quando ele está ativado na função *reset*.
  3. Quando ativo, o módulo de supervisão de alimentação (SVS) detecta uma queda da tensão de alimentação (para mais detalhes, veja o tópico 5.20).
- **PUC** (***Power Up Clear*** - algo como confirmação de alimentação estável) - este é um sinal secundário de *reset*, gerado por um dos seguintes eventos:
  1. POR ativo: um evento de *power-on reset* dispara um sinal PUC.
  2. Um estouro da contagem do *watchdog* (cão de guarda), quando ele está configurado para o modo *reset*.

3. Uma violação na senha de acesso ao registrador de controle do *watchdog*. A tentativa de alterar o registrador de controle do *watchdog* sem a escrita do valor (senha) correto provoca um *reset* PUC (veja o tópico 5.21.6).
4. Uma violação de acesso aos registradores de controle do controlador da FLASH. A tentativa de alteração do registrador do controlador de memória FLASH sem o código (senha) correto provoca um *reset* PUC (veja tópico 5.21).
5. A busca de instrução num endereço pertencente à área de registradores (0x0000 a 0x01FF) provoca um *reset* PUC (somente nos *chips* da família 2xx).

5.1.1. Sistema BOR

Alguns modelos da família MSP430 possuem um circuito de POR mais elaborado que se chama BOR (***Brown-Out Reset*** - algo como *reset* por distúrbio da alimentação).

O circuito BOR diferencia-se do POR porque inclui uma histerese, ou seja, há uma margem intermediária composta de dois valores: $V_{(B_IT+)}$ e $V_{(B_IT-)}$. Quando o sistema é inicialmente alimentado e a tensão de alimentação começa a subir, o circuito BOR manterá o sistema em *reset* até que a tensão ultrapasse a margem superior ($V_{(B_IT+)}$). Cerca de 2ms após, o BOR desativa o sinal de *reset* e a CPU inicia a execução do programa.

Caso a tensão de alimentação comece a diminuir, seja por uma interrupção na alimentação, seja pela descarga de uma bateria que esteja servindo de fonte, o circuito BOR gera um sinal de *reset* quando ela cair abaixo da margem inferior ($V_{(B_IT-)}$) e mantém a CPU em *reset* até que a tensão de alimentação volte a ultrapassar a margem superior ($V_{(B_IT+)}$).

Uma característica importante do BOR é o seu consumo extremamente baixo, da ordem de nA, permitindo o seu uso em aplicações alimentadas por baterias.

Os modelos de MSP430 que incluem o circuito BOR são: MSP430x1122, 1132, 1222, 1232, 15x, 16x e todos os modelos das famílias 2xx e 4xx.

5.1.2. Efeitos do *Reset*

Após um *reset* POR ou BOR, a CPU e alguns registradores SFR são inicializados em uma condição-padrão, conforme a seguir:

1. O pino $\overline{\text{RST}}$/NMI é configurado para o modo *reset*.
2. As portas de E/S são configuradas para a função de entrada digital.
3. O registrador de *status* (SR) é apagado (todos os *bits* em nível lógico "0").
4. **O *watchdog* é ativado no modo *reset*.**
5. O contador de programa é carregado com o conteúdo indicado pelo vetor de *reset* (o *word* contida no endereço 0xFFFE).

Repare que o *reset* não realiza o ajuste do apontador do topo da pilha (SP). Essa tarefa deve ser providenciada pelo *software* do usuário.

5.2. Sistema de Interrupções

Uma interrupção consiste em um evento externo ao programa que provoca um desvio no seu fluxo, de forma que a CPU passa executar um subprograma em resposta ao evento. Ao término desse subprograma (chamado normalmente de ISR (*Interrupt Servicing Routine*) ou RTI (**R**otina de **T**ratamento de **I**nterrupção)), o fluxo do programa retorna ao ponto em que se encontrava antes da interrupção.

Interrupções são úteis quando necessitamos que a CPU responda rapidamente a um evento, mas sem a perda de capacidade de executar outras operações enquanto ele não ocorra.

O tempo decorrido entre o início do evento e o início da execução da RTI é chamado de latência de interrupção e quanto menor, mais rapidamente a CPU responde ao evento.

No caso dos MSP430, a latência de interrupção é fixa e determinada. São necessários seis ciclos de *clock* para que a CPU reconheça a interrupção e efetue todo o procedimento de desvio do fluxo do programa.

Como resposta a um evento de interrupção, a CPU executa os seguintes procedimentos:

1. A instrução em execução é completada.
2. O conteúdo do contador de programa (PC) é salvo na pilha, de forma que o programa possa posteriormente retornar ao ponto em que se encontrava.
3. O conteúdo do registrador SR é salvo na pilha, de forma a preservar os *bits* de modo de operação e *flags* matemáticos da CPU.
4. Caso existam múltiplas interrupções simultâneas, aquela com mais alta prioridade é tratada primeiro.
5. O *flag* da interrupção é apagado automaticamente pela CPU. Em alguns tipos de interrupção, em que um *flag* é ativado por mais de um tipo de interrupção, o *flag* deve ser apagado pelo *software* do usuário. Essa peculiaridade será vista mais adiante.
6. O conteúdo do registrador SR é apagado, com exceção do *bit* SCG0. Isso faz com que o *chip* saia de um eventual modo de baixo consumo em que se encontra. Outro efeito do apagamento do SR é que o *bit* GIE é também apagado, impedindo que novas interrupções não-mascaráveis ocorram durante o tratamento da atual.
7. O vetor correspondente à interrupção acionada é carregado no PC, o que provoca o desvio do fluxo do programa para a RTI correspondente.

5.2.1. Categorias de Interrupção

Existem duas categorias básicas de interrupção nos MSP430: não-mascaráveis e mascaráveis. A única diferença entre elas reside no fato de que as primeiras não podem ser desativadas pelo controle geral de interrupções (o *bit* GIE do SR), enquanto as mascaráveis dependem de que o *bit* GIE esteja ativado (nível "1") para que possam ser reconhecidas pelo controlador de interrupções da CPU.

Além das duas categorias anteriores, ainda temos um terceiro tipo de interrupção: o *reset*. Nos MSP430 o *reset* é tratado como uma categoria especial de interrupção não-mascarável, possuindo prioridade máxima e um vetor próprio (0xFFFE).

São três os eventos capazes de gerar uma interrupção não-mascarável:

- Uma transição de nível no pino $\overline{\text{RST}}$/NMI quando ele está configurado no modo NMI. Para maiores detalhes sobre o funcionamento desse pino e da interrupção NMI veja o tópico 5.23.
- Uma falha no oscilador: caso um dos osciladores (LFXT1 ou XT2) deixe de funcionar, o *flag* OFIFG é ativado, sinalizando a falha no oscilador. Caso a interrupção esteja habilitada, o programa será desviado para o endereço apontado pelo vetor 14. Maiores detalhes serão vistos no estudo do módulo oscilador (tópico 5.3).
- Uma violação de acesso à memória *FLASH* (tópico 5.21 sobre o controlador da memória *FLASH*).

Observe que cada interrupção descrita pode ser individualmente habilitada/desabilitada pelo *software* do usuário.

Um detalhe de extrema importância sobre esta categoria de interrupções: após a ocorrência de uma NMI, os seus respectivos controles de habilitação de interrupção (OFIE, NMIIE e ACCVIE) são automaticamente apagados pelo *hardware* do microcontrolador, após efetuado o desvio para a rotina de tratamento de interrupção.

Caso uma ou mais interrupções deste tipo estejam sendo utilizadas na aplicação, é importante que as mesmas sejam novamente habilitadas dentro da própria RTI. O fabricante recomenda que esta operação seja feita pouco antes da execução da instrução de retorno de interrupção e que os *flags* sejam habilitados em uma única operação de escrita, de forma a evitar que uma interrupção NMI possa acontecer antes do término do tratamento da NMI corrente, o que acarretaria em um aninhamento de interrupções, que por sua vez poderia levar a uma condição conhecida como "estouro da pilha", em inglês *stack overflow*.

5.2.2. Vetores de Interrupção

A arquitetura MSP430 inclui um sistema de controle de interrupções dotado de 16 diferentes vetores, cada um deles possui um grau de prioridade, sendo o vetor 0 o de menor prioridade e o vetor 15 o de maior prioridade.

Cada vetor armazena o endereço em que está localizada a sub-rotina responsável pelo tratamento do evento especificado.

Vetor	Endereço	Classe	Fonte	*Flag*	Registrador	Modelos
15	0xFFFE	*Reset*	POR/BOR Pino RST	-	-	Todos
			Watchdog	WDTIFG	IFG1	
			Memória FLASH	KEYV	FCTL3	
14	0xFFFC	Interrupção não-mascarável	Pino NMI	NMIIFG	IFG1	Todos
			Falha do oscilador	OFIFG	IFG1	
			Violação de acesso à memória FLASH	ACCVIFG	FCTL3	

Vetor	Endereço	Classe	Fonte	*Flag*	Registrador	Modelos
13	0xFFFA	Interrupção mascarável	Timer B - canal 0	CCIFG	TBCCTL0	13x, 14x, 15x, 16x, 43x, 44x
			Timer 1 - canal 0	CCIFG	TA1CCTL0	415, 417, FW42x
			ESP430	OUT0IFG OUT1IFG OUT0FG OUT1FG IN0IFG IN1IFG	MBCTL	FE42x
12	0xFFF8	Interrupção mascarável	Timer B - canais 1 e 2	CCIFG	TBCCTL1 e TBCCTL2	13x, 15x, 43x
			Timer B - canais 1 a 6	CCIFG	TBCCTL1 a TBCCTL6	14x, 16x, 44x
			Timer B - estouro de contagem	TBIFG	TBCTL	13x, 14x, 15x, 16x, 43x, 44x
			Timer 1 - canais 1 a 4	CCIFG	TA1CCR1 a TA1CCR4	FW42x
			Timer 1 - estouro de contagem	TAIFG	TA1CTL	
			SD16 - *overflow* ou fim de conversão	OVIFG IFG	SD16CCTLx	FE42x
11	0xFFF6	Interrupção mascarável	Comparador A	CAIFG	CACTL1	1101, 1111, 1121, 122, 123, 13x, 14x, 15x, 16x, 412, 413, 415, 417, FW42x, 43x, 44x
10	0xFFF4	Interrupção mascarável	Estouro do *watchdog*	WDTIFG	IFG1	Todos
9	0xFFF2	Interrupção mascarável	Recepção na USART0	URXIFG0	IFG1	13x, 14x, 15x, 16x, 42x, FE42x, 43x, 44x
			SCAN IF	SIFIFGx	SIFCTL1	FW42x
			Timer A - canal 0	CCIFG	TACCTL0	11xx, 12xx
8	0xFFF0	Interrupção mascarável	Transmissão USART0	UTXIFG0	IFG1	13x, 14x, 15x, 16x, 42x, FE42x, 43x, 44x
			I2C	STTIFG GCIFG TXRDYIFG RXRDYIFG ARDYIFG OAIFG NACKIFG ALIFG	I2CIFG	15x, 16x
			Timer A - canais 1 e 2	CCIFG	TACCTL1 e TACCTL2	11xx, 12xx
			Timer A - estouro de contagem	TAIFG	TACTL	
7	0xFFEE	Interrupção mascarável	ADC12 - fim de conversão do canal x	ADC12IFGx	ADC12IFG	133, 135, 147, 148, 149, 15x, 16x, 43x, 44x
			Recepção USART0	URXIFG0	IFG1	12xx
6	0xFFEC	Interrupção mascarável	Timer A - canal 0	CCIFG	TACCTL0	13x, 14x, 15x, 16x, 41x, 42x, 43x, 44x
			Timer 0 - canal 0	CCIFG	TA0CCTL0	FW42x
			Transmissão USART0	UTXIFG0	IFG1	12xx

Vetor	Endereço	Classe	Fonte	*Flag*	Registrador	Modelos
5	0xFFEA	Interrupção mascarável	Timer A - canal 1 e 2	CCIFG	TACCTL1 e TACCTL2	13x, 14x, 15x, 16x, 41x, 42x, 43x, 44x
			Timer A - estouro de contagem	TAIFG	TACTL	
			Timer 0 - canal 1 e 2	CCIFG	TA0CCTL1 e TA0CCTL2	FW42x
			ADC10 - fim de conversão	ADC10IFG	ADC10CTL0 e ADC10CTL1	11x2, 12x2
4	0xFFE8	Interrupção mascarável	Porta 1 - mudança de estado do pino	P1IFG.0 a P1IFG.7	P1IFG	13x, 14x, 15x, 16x, 4xx
3	0xFFE6	Interrupção mascarável	Porta 2 - mudança de estado do pino	P2IFG.0 a P2IFG.7	P2IFG	11xx, 12xx
			Recepção USART1	URXIFG1	IFG2	14x, 15x, 16x, 44x
			DAC12	DAC12IFG	DAC12_1CTL e DAC12_2CTL	FG43x
			DMA - fim da transferência	DMAIFG	DMA0CTL	FG43x
2	0xFFE4	Interrupção mascarável	Porta 1 - mudança de estado do pino	P1IFG.0 a P1IFG.7	P1IFG	11xx, 12xx
			Transmissão USART1	UTXIFG1	IFG2	14x, 15x, 16x, 44x
1	0xFFE2	Interrupção mascarável	Porta 2 - mudança de estado do pino	P2IFG.0 a P2IFG.7	P2IFG	13x, 14x, 15x, 16x, 4xx
0	0xFFE0	Interrupção mascarável	DMA	DMAIFG	DMA0CTL, DMA1CTL e DMA2CTL	15x, 16x
			DAC12	DAC12IFG	DAC12_1CTL e DAC12_2CTL	15x, 16x
			Timer básico - estouro de contagem	BTIFG	IFG2	4xx

Tabela 5-1

Repare que, apesar da existência de apenas 16 vetores, a arquitetura não está restrita a apenas 16 interrupções diferentes. Vários módulos possuem *flags* que especificam o evento interno originador da interrupção. Isso permite ampliar a quantidade de diferentes eventos de interrupção tratáveis pelo *software* do usuário.

A título de exemplo, podemos destacar o vetor 14 que é compartilhado por três diferentes módulos de *hardware*. Cada um com o seu *flag* individual. A RTI deve providenciar a verificação de cada um dos *flags* (NMIIFG, OFIFG e ACCVIFG) de forma a identificar o evento causador da interrupção.

Também há periféricos que possuem um "gerador de vetor de interrupção" que consiste em um registrador cujo conteúdo especifica qual das fontes de interrupção internas ao periférico foi a causadora da interrupção.

Esse tipo de abordagem, apesar de aumentar ligeiramente a complexidade do *software* de tratamento de interrupção, permite a construção de uma arquitetura de interrupções bastante simples e eficiente.

5.2.3. Registradores de Controle de Interrupção

A arquitetura MSP430 prevê dois tipos básicos de registradores de controle de interrupção, todos localizados na área de registradores SFR: os registradores de habilitação de interrupção (IEx) e os registradores sinalizadores de interrupção (IFGx). Além deles, muitos *flags* de interrupção encontram-se localizados nos registradores de controle dos periféricos (como podemos observar pela coluna registrador na tabela 5-1).

5.2.3.1. IE1

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0000	IE1	Leitura / Escrita	UTXIE0	URXIE0	ACCVIE	NMIIE	-	-	OFIE	WDTIE
		Reset	0	0	0	0	0	0	0	0

UTXIE0 - habilitação da interrupção por transmissão da USART0 (símbolo **UTXIE0**);

URXIE0 - habilitação da interrupção por recepção da USART0 (símbolo **URXIE0**);

ACCVIE - habilitação da interrupção por violação de acesso à memória *FLASH* (símbolo **ACCVIE**);

NMIIE - habilitação da interrupção não-mascarável (NMI) (símbolo **NMIIE**);

OFIE - habilitação da interrupção por falha no oscilador (símbolo **OFIE**);

WDTIE - habilitação da interrupção por estouro da contagem do *watchdog* (quando ele está operando no modo *watchdog*) (símbolo **WDTIE**);

Bit = 1 - interrupção habilitada;

Bit = 0 - interrupção desabilitada.

5.2.3.2. IE2

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0001	IE2	Leitura / Escrita	BTIE	-	UTXIE1	URXIE1	-	-	UTXIE0*	URXIE0*
		Reset	0	0	0	0	0	0	0	0

* Somente nos dispositivos 12xx.

BTIE - habilitação da interrupção do *timer* básico 1 (somente na família 4xx) (símbolo **BTIE**);

UTXIE1 - habilitação da interrupção de transmissão da USART1 (símbolo **UTXIE1**);

URXIE1 - habilitação da interrupção de recepção da USART1 (símbolo **URXIE1**);

UTXIE0 - habilitação da interrupção por transmissão da USART0 (somente nos modelos 12xx) (símbolo **UTXIE0**);

URXIE0 - habilitação da interrupção por recepção da USART0 USART0 (somente nos modelos 12xx) (símbolo **URXIE0**);

Bit = 1 - interrupção habilitada;

Bit = 0 - interrupção desabilitada.

5.2.3.3. IFG1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0002	IFG1	Leitura / Escrita	UTXIFG0	URXIFG0	-	NMIIFG	RSTIFG	PORIFG	OFIFG	WDTIFG
		Reset	1	0	0	0	0	0	0	0

UTXIFG0 - sinalizador da interrupção de transmissão da USART0 (símbolo **UTXIFG0**);
URXIFG0 - sinalizador da interrupção de recepção da USART0 (símbolo **URXIFG0**);
NMIIFG - sinalizador da interrupção não-mascarável (símbolo **NMIIFG**);
RSTIFG - sinalizador de *reset* externo (pino RST) (somente na família 2xx) (símbolo **RSTIFG**);
PORIFG - sinalizador de *power-on* (somente na família 2xx) (símbolo **PORIFG**);
OFIFG - sinalizador de interrupção por falha no oscilador (símbolo **OFIFG**);
WDTIFG - sinalizador de interrupção por estouro de contagem do *watchdog* (quando operando no modo *watchdog*) (símbolo **WDTIFG**).

5.2.3.4. IFG2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0003	IFG2	Leitura / Escrita	BTIFG	-	UTXIFG1	URXIFG1	-	-	UTXIFG0*	URXIFG0*
		Reset	0	0	1	0	0	0	1	0

* Somente nos dispositivos 12xx.

BTIFG - sinalizador da interrupção *timer* básico 1 (somente na família 4xx) (símbolo **BTIFG**);
UTXIFG1 - sinalizador da interrupção de transmissão da USART1 (símbolo **UTXIFG1**);
URXIFG1 - sinalizador da interrupção de recepção da USART1 (símbolo **URXIFG1**);
UTXIFG0 - sinalizador da interrupção de transmissão da USART0 (somente nos modelos 12xx) (símbolo **UTXIFG0**);
URXIFG0 - sinalizador da interrupção de recepção da USART0 (somente nos modelos 12xx) (símbolo **URXIFG0**).

5.2.4. Tratamento de Interrupções

Para terminarmos o estudo sobre o funcionamento das interrupções nos MSP430, resta-nos falar sobre os procedimentos e códigos necessários ao tratamento das interrupções.

Para fins didáticos, vamos implementar duas aplicações, uma em *Assembly* e outra em C, com o objetivo de piscar um LED conectado ao pino P1.0. Nestes exemplos, vamos configurar o *timer* A para gerar uma interrupção periódica. A RTI do *timer* A será a responsável por inverter o estado do LED.

```
#include <msp430x16x.h>

        NAME    main
        PUBLIC  main
        ORG     0xFFEA      ; vetor da interrupção TAIFG
        DC16    trata_timer ; define o endereço da RTI
        ORG     0FFFEh
        DC16    main
        RSEG    CODE

main    MOV     #0x3FF,SP   ; inicializa o apontador da pilha
        MOV     #WDTPW+WDTHOLD,WDTCTL ; desliga o watchdog
        MOV     #TASSEL_1+ID_3+MC_1+TAIE,TACTL  ; configura o timer
        MOV     #0x8000,TACCR0        ; configura o módulo de contagem
        MOV.B   #1,P1DIR              ; configura P1.0 como saída
        EINT                          ; habilita interrupções
fim     JMP     fim                   ; loop infinito

trata_timer                           ; RTI
        XOR.B   #1,P1OUT              ; inverte o estado do pino P1.0
        RETI                          ; retorna da interrupção

        END     main
```

Exemplo 5-1

Como podemos ver, em *Assembly*, utilizamos a diretiva ORG para definir o endereço do vetor de interrupção desejado (no caso 0xFFEA) e em seguida, definimos uma constante que será montada naquele endereço por meio da diretiva DC16. A constante é o endereço da sub-rotina "trata_timer".

Observe que no exemplo 5-1 não houve cuidado com o salvamento do contexto na entrada da RTI nem com a restauração do contexto na sua saída. Neste caso em particular, isso não foi necessário, uma vez que o programa principal não executa nenhum código enquanto aguarda a interrupção.

Se houvesse código sendo executado no *loop* principal do programa, seria necessário salvar o estado dos registradores afetados pela execução da RTI. No próximo exemplo, temos um contador baseado no registrador R9 no *loop* principal. Esse registrador também é apagado pela RTI (simulando a sua utilização), por isso foram incluídas instruções PUSH e POP para o salvamento e recuperação do valor de R9 utilizando a pilha.

A seqüência de salvamento dos registradores deve ser respeitada no instante da restauração dos valores, ou seja, o último registrador salvo deve ser o primeiro a ser recuperado e assim por diante, até que o primeiro registrador salvo seja recuperado (lembre-se de que a pilha é uma estrutura LIFO (o último a entrar é o primeiro a sair)).

```
#include <msp430x16x.h>

        NAME    main
        PUBLIC  main
        ORG     0xFFEA      ; vetor da interrupção TAIFG
        DC16    trata_timer ; define o endereço da RTI
        ORG     0FFFEh
        DC16    main
        RSEG    CODE
```

```
main    MOV     #0x3FF,SP    ; inicializa o apontador da pilha
        MOV     #WDTPW+WDTHOLD,WDTCTL ; desliga o watchdog
        MOV     #TASSEL_1+ID_3+MC_1+TAIE,TACTL  ; configura o timer
        MOV     #0x8000,TACCR0        ; configura o módulo de contagem
        MOV.B   #1,P1DIR              ; configura P1.0 como saída
        EINT                          ; habilita interrupções
        CLR     R9                    ; apaga o R9
loop    INC     R9                    ; incrementa o contador
        BR      #loop                 ; loop infinito

trata_timer                           ; RTI
        PUSH    R9                    ; salva o estado de R9 na pilha
        CLR     R9                    ; apaga R9
        XOR.B   #1,P1OUT              ; inverte o estado do pino P1.0
        POP     R9                    ; restaura o estado de R9 da pilha
        RETI                          ; retorna da interrupção

        END     main
```

Exemplo 5-2

Para realizarmos o tratamento de interrupções em C, utilizamos a diretiva **#pragma vector** = que informa ao compilador que a próxima função deve ter o seu endereço de entrada armazenado na tabela de vetores de interrupção, na posição indicada após o sinal de igual.

A palavra reservada **__interrupt** informa ao compilador que a função definida por ela é uma função de tratamento de interrupção. Isso faz com que o compilador gere os códigos de salvamento de registradores na entrada da função e de retorno de interrupção na sua saída.

Também é possível utilizar adicionalmente a palavra reservada **__raw**, que faz com que o compilador não gere código para salvamento dos registradores da CPU na entrada e de restauração na saída da função de interrupção. Neste caso, o programador deve providenciar, se for necessário, todo o código de salvamento e recuperação de contexto.

```
#include <io430x16x.h>
#include <intrinsics.h>

#pragma vector = TIMERA1_VECTOR
__interrupt void trata_timer(void)
{
   // inverte o estado do P1.0
   P1OUT^=1;
}

int main( void )
{
  WDTCTL = WDTPW + WDTHOLD;          // desativa o watchdog
  P1DIR_bit.P1DIR_0 = 1;             // configura o pino P1.0 como saída
  TACTL = TASSEL_1+ID_3+MC_1+TAIE;   // contigura o timer
  TACCR0 = 0x8000;                   // configura o módulo de contagem
  __bis_SR_register(GIE);            // habilita as interrupções
  while(1);                          // loop infinito
}
```

Exemplo 5-3

A tabela 5-2 apresenta os símbolos utilizados para os diversos vetores de interrupção disponíveis na família MSP430.

Vetor	Símbolo
Reset	RESET_VECTOR
NMI	NMI_VECTOR
Porta 1	PORT1_VECTOR
Porta 2	PORT2_VECTOR
Timer A, CCP0	TIMERA0_VECTOR
Timer A, demais interrupções	TIMERA1_VECTOR
Timer B, CCP0	TIMERB0_VECTOR
Timer B, demais interrupções	TIMERB1_VECTOR
Timer Básico	BASICTIMER_VECTOR
USART0_TX	USART0TX_VECTOR
USART0_RX	USART0RX_VECTOR
USART1_TX	USART1TX_VECTOR
USART1_RX	USART1RX_VECTOR
Watchdog	WDT_VECTOR
Comparador A	COMPARATORA_VECTOR
ADC10 e ADC12	ADC_VECTOR
ADC16	SD16_VECTOR
DAC/DMA (família 1xx)	DACDMA_VECTOR
DAC/DMA (família FG43x)	DAC12_DMA_VECTOR
ESP430	ESP430_VECTOR
SCANIF	SCANIF_VECTOR

Tabela 5-2

Lembre-se de que, para habilitar as interrupções num programa em C, podemos utilizar (para utilizar estas funções, é necessário incluir o arquivo intrinsics.h):

```
__bis_SR_register(GIE);
```

Ou:

```
__enable_interrupt();
```

Para desabilitar as interrupções, podemos utilizar:

```
__bic_SR_register(GIE);
```

Ou,

```
__disable_interrupt();
```

5.3. Módulo Oscilador

Os microcontroladores MSP430 incluem um avançado sistema de *clock*, que permite a operação da CPU e dos periféricos a partir de diferentes fontes.

Nos dispositivos da família 1xx, o sistema de *clock* é chamado de BCS (***Basic Clock System*** ou sistema básico de *clock*) e é composto de um ou dois osciladores (dependendo do modelo do *chip*) capazes de funcionar com cristais ou ressonadores externos, além de um oscilador interno controlado digitalmente (DCO).

Nos dispositivos da família 2xx, o sistema de *clock* é chamado de BCS+ e é praticamente idêntico ao BCS, com algumas inovações como a ampliação da freqüência de trabalho (até 16 MHz), diminuição do consumo de energia e diminuição do tempo de partida do oscilador interno.

Nos dispositivos da família 4xx, o sistema de *clock* é chamado de FLL+ (***Frequency-Locked Loop*** ou laço amarrado em freqüência). Esse sistema é composto de um ou dois osciladores capazes de funcionar com cristais ou ressonadores externos e um oscilador interno controlado digitalmente (DCO), que nesse sistema é ajustado e controlado por um *hardware* interno, de forma a garantir que ele trabalhe em uma freqüência múltipla de um cristal externo de baixa freqüência.

Os sinais provenientes desses osciladores podem ser selecionados para a geração de três sinais de *clock* do sistema: o sinal de *clock* principal (MCLK), o sinal de *clock* secundário (SMCLK) e o sinal auxiliar de *clock* (ACLK). Cada um desses sinais pode ser dividido internamente por um fator de 1, 2, 4 ou 8, antes de estar disponível para a CPU e os periféricos.

O *clock* principal (MCLK) é basicamente utilizado para sincronizar a CPU e eventualmente outros periféricos do sistema.

O *clock* secundário (SMCLK) é utilizado basicamente como fonte de *clock* alternativa para os diversos periféricos do microcontrolador.

O *clock* auxiliar (ACLK) possui a finalidade básica de funcionar como fonte de *clock* de precisão para periféricos durante modos de baixo consumo.

5.3.1. Oscilador de Baixa/Alta Freqüência (LFXT1)

O oscilador LFXT1 é um circuito encontrado em todos os dispositivos MSP430, utilizado basicamente para fornecer sinais de *clock* de precisão para periféricos como os *timers*. Esse oscilador pode ser mantido em operação na maioria dos modos de baixa potência, permitindo que o projetista mantenha os periféricos internos em operação, enquanto a CPU encontra-se parada.

O oscilador LFXT1 pode trabalhar em dois modos distintos: baixa freqüência, em que normalmente utiliza um cristal de 32768Hz ou alta freqüência, com cristais ou ressonadores cerâmicos de 450KHz a 8MHz (16MHz na família 2xx). A seleção do modo de funcionamento é feita pelo *bit* XTS (0 = baixa freqüência (LF), 1 = alta freqüência (XT)), localizado no registrador BCSCTL1 (famílias 1xx e 2xx) ou pelo *bit* XTS_FLL, localizado no registrador FLL_CTL0 (família 4xx).

Nas figuras 5-1 e 5-2 temos o diagrama em blocos desse oscilador para as famílias 1xx e 4xx respectivamente. A diferença básica entre os dois, fora a nomenclatura do sinal ACLK, é que nos dispositivos da família 1xx o oscilador apresenta capacitores internos fixos (2pF no modo LF e 12pF no modo HF). No caso dos dispositivos das famílias 2xx e 4xx, esses capacitores são selecionáveis com valores de 1, 6, 8 ou 10pF.

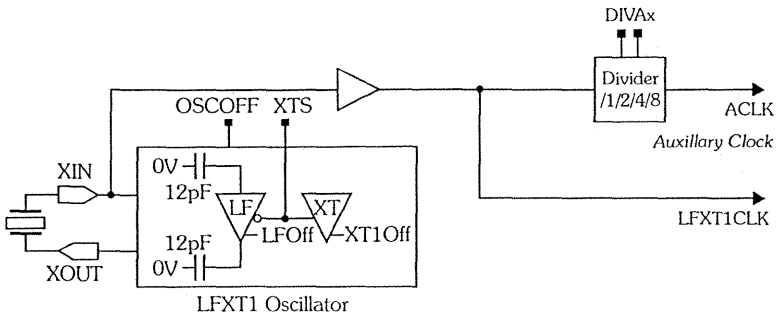


Figura 5-1

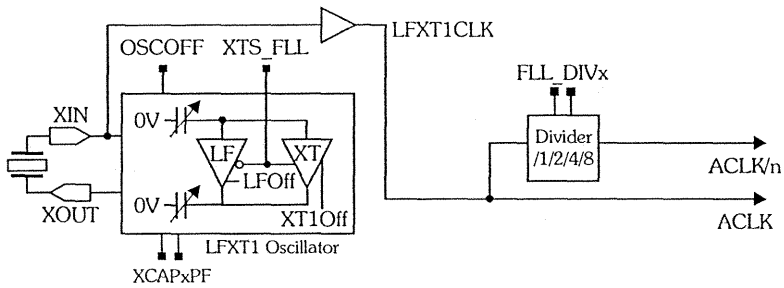


Figura 5-2

Repare que o sinal LFXT1CLK gerado pelo oscilador, após ser dividido por um fator de 1, 2, 4 ou 8, origina o sinal de *clock* auxiliar do sistema (ACLK nas famílias 1xx e 2xx e ACLK/n na família 4xx). O sinal de *clock* auxiliar ACLK pode ser utilizado como fonte de *clock* para praticamente qualquer periférico do MSP430.

A principal aplicação do oscilador LFXT1 é sem dúvida a construção de relógios de tempo real (RTC). Neste caso, utilizam-se o modo de baixa freqüência e um cristal de 32768 Hz. Configurando um *timer* para operar a partir do sinal ACLK, podemos manter o *chip* em um modo de consumo muito baixo (utilizando, por exemplo, o modo LPM3), ao mesmo tempo em que se mantém um relógio baseado em *timer* funcionando. A cada intervalo de tempo definido o *timer* gera uma interrupção, fazendo com que o *chip* saia do modo de baixo consumo e execute a rotina de tratamento de interrupção, que deve atualizar o horário e se for o caso, retornar ao modo de baixo consumo, garantindo desta forma uma grande economia de energia.

Nos dispositivos dotados de controlador de LCD, o sinal ACLK também pode ser utilizado para a atualização do controlador, permitindo que ele continue funcionando, mesmo com o *chip* em um modo de baixa potência.

Algumas observações importantes acerca do oscilador LFXT1:

- O oscilador pode ser desligado (*bit* OSCOFF = 1), caso não esteja sendo utilizado, de forma a reduzir o consumo de energia.
- É possível utilizar um sinal de *clock* proveniente de um oscilador externo. Neste caso, o sinal deve ser aplicado ao pino XIN.
- No modo de baixa freqüência, é importante colocar o cristal o mais próximo possível do microcontrolador. Uma outra prática recomendada é cercar as trilhas de conexão ao cristal por uma malha de terra. Também é interessante conectar a carcaça do cristal à malha de terra.

- Quando opera no modo de baixa freqüência e com tensões de alimentação inferiores a 2,5V, é necessário utilizar um resistor de 5,1MΩ entre o pino XOUT e o terra do circuito.
- Lembre-se de que um oscilador de baixa freqüência pode necessitar de muito tempo (da ordem de centenas de milissegundos) para iniciar corretamente a sua oscilação. Nestes casos, é interessante não desligar o oscilador ao entrar em um modo de baixo consumo.
- Nos microcontroladores da família 2xx, os pinos XIN e XOUT são multiplexados com os bits 6 e 7 da porta 2. Caso o oscilador LFXT1 não esteja sendo utilizado, esses pinos podem ser utilizados para E/S (apagando os bits 6 e 7 do registrador P2SEL).
- Na família 4xx, o sinal ACLK possui sempre a mesma freqüência do sinal LFXT1CLK. O sinal ACLK/n somente pode ser utilizado externamente e não está disponível para os periféricos.

5.3.2. Oscilador de Alta Freqüência (XT2)

Alguns dispositivos das famílias 1xx, 2xx e 4xx implementam um segundo oscilador, projetado especificamente para operar em altas freqüências (450kHz a 8MHz, ou 16MHz no caso da família 2xx). Esse oscilador comporta-se exatamente como o oscilador LFXT1 no modo de alta freqüência (XT).

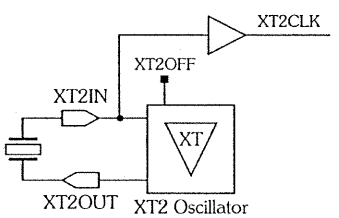


Figura 5-3

Assim como o LFXT1, o XT2 também apresenta capacitores internos de 2pF. Dispensando, na maioria das vezes, o uso de capacitores externos.

O oscilador XT2 também pode ser desativado pelo *bit* XT2OFF (registrador BCSCTL1 nas famílias 1xx e 2xx, registrador FLL_CTL1 na família 4xx), desde que não esteja fornecendo *clock* para o MCLK ou SMCLK.

5.3.3. DCO

Os dispositivos das famílias 1xx e 2xx incluem um circuito oscilador controlado digitalmente (DCO), capaz de operar em uma gama bastante ampla de freqüências e sob o controle do usuário.

O DCO consiste basicamente em um oscilador RC, em que o elemento resistivo (R) pode ser selecionado pelo usuário entre duas opções possíveis, sendo uma interna (integrada na pastilha do MCU e com uma resistência de aproximadamente 200KΩ) e uma externa (um resistor conectado externamente ao pino R_{OSC}). A seleção é feita pelo *bit* DCOR (registrador BCSCTL2).

O sinal de *clock* proveniente do gerador DC é aplicado a um modulador cuja função é misturar dois sinais de freqüências diferentes, de forma a obter um terceiro sinal, com freqüência entre os dois.

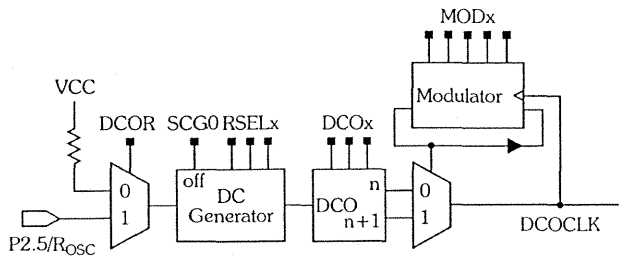


Figura 5-4

O funcionamento do modulador é relativamente simples. O sinal de *clock* obtido no gerador DC é aplicado a um divisor programável que fornece oito sinais de saída, cada um cerca de 10% superior ao anterior. O sinal proveniente da saída selecionada pelo usuário por meio dos *bits* DCOx (registrador DCOCTL) e o sinal imediatamente superior (cerca de 10% superior ao selecionado pelo usuário) são aplicados a um multiplexador de duas entradas, que seleciona um dos dois de acordo com a contagem feita pelo modulador. Ou seja, o fator programado no modulador (MODx, registrador DCOCTL) determina quantas vezes em 32 contagens será utilizado o sinal de maior freqüência.

Cada unidade aumentada no fator MODx resulta em um aumento de 1/32 da freqüência base selecionada pelo fator DCOx. Com MODx = 0 temos um *clock* igual ao fator DCOx selecionado e com MODx = 31 temos um *clock* aproximadamente igual à cerca de 96% do fator imediatamente superior.

A fórmula geral para o cálculo da freqüência de saída do modulador é:

$$f = \frac{(32 - MODx) * f_{DCO} + MODx * f_{DCO+1}}{32}$$

Observe que, apesar de a mistura de duas freqüências diferentes resultar efetivamente em uma variação na freqüência de *clock*, essa variação não introduz erros cumulativos, uma vez que a cada 32 ciclos de *clock* o processo se repete.

Na tabela 5-3 temos os valores de freqüência típicos para as diversas seleções de resistores internos e fatores de divisão do DCO (válidos para a família 1xx com MODx = 0):

RSELx	DCOx							
	0	1	2	3	4	5	6	7
0	88,8KHz	97,3KHz	106,4KHz	116,9KHz	128,7KHz	141,9KHz	156,5KHz	176,8KHz
1	140,9KHz	154,5KHz	168,9KHz	185,4KHz	203,8KHz	224,6KHz	247,3KHz	279,7KHz
2	221,2KHz	242,3KHz	264,8KHz	290,9KHz	319,7KHz	352,3KHz	387,5KHz	439,1KHz
3	370,3KHz	405,3KHz	442,9KHz	486,5KHz	534,8KHz	589,8KHz	649,5KHz	736,9KHz
4	625,7KHz	685,1KHz	748,9KHz	823KHz	904,7KHz	998,9KHz	1,1MHz	1,25MHz
5	1,04MHz	1,13MHz	1,24MHz	1,36MHz	1,49MHz	1,66MHz	1,83MHz	2,08MHz
6	1,64MHz	1,79MHz	1,96MHz	2,16MHz	2,38MHz	2,64MHz	2,92MHz	3,31MHz
7	2,56MHz	2,79MHz	3,06MHz	3,37MHz	3,72MHz	4,13MHz	4,59MHz	5,19MHz

Tabela 5-3

Repare que, no caso da família 2xx, existem 16 seleções diferentes para o resistor interno do DCO (RSELx pode variar de 0 a 15).

A tabela 5-4 apresenta os valores de freqüência típicos para alguns resistores-padrão conectados ao pino R_{OSC} (com o fator DCOx = 7 e RSELx = 7, família 1xx). Para utilizar o DCO com um resistor externo, basta setar o *bit* DCOR (registrador BCSCTL2). Observe que os valores da tabela são aproximados e válidos para a família 1xx.

Resistência (Ohms)	Freqüência (MHz)
1K	17,6
10K	16,44
47K	12,77
100K	9,89
150K	8,08
220K	6,29
470K	3,45

Tabela 5-4

Observe que a seleção RSELx possui também influência na freqüência de oscilação quando operando com resistor externo, porque os *bits* RSELx atuam sobre o gerador DC do DCO.

5.3.3.1. Observações Importantes sobre o DCO

- Quando opera com o resistor interno, o DCO pode apresentar desvios de freqüência devido a fatores como a temperatura ambiente e a tensão de alimentação do *chip*. Uma alternativa para reduzir a dependência da temperatura é utilizar um resistor externo conectado ao pino R_{OSC} (normalmente multiplexado com o P2.5).
- Na família 2xx, em virtude do aumento da estabilidade em temperatura do DCO, não existe a opção de utilização de um resistor externo para controlá-lo.
- A utilização de um resistor externo deve respeitar a freqüência máxima de operação especificada pelo fabricante (8MHz para a família 1xx e 16MHz para a família 2xx). A operação em freqüências superiores às especificadas pelo fabricante, apesar de possível, não é recomendada.
- O DCO é a fonte-padrão de *clock* quando o MSP sai de um *reset*. Nessa condição, RSELx = 4 (registrador BCSCTL1), DCOx = 3 e MODx = 0, fazendo com que o *chip* trabalhe a aproximadamente 823KHz. Uma vez que o DCO possui um tempo de partida muito rápido (da ordem de 6μs), a CPU pode responder rapidamente a um evento de saída de um *reset* ou de um modo de baixa potência.
- O DCO pode ser desligado, caso não se pretenda utilizá-lo. O *bit* SCG0 do registrador SR, quando setado, desliga o gerador DC do DCO, reduzindo o consumo de energia do MCU.

5.3.4. FLL

Os dispositivos da família 4xx incluem um circuito FLL, capaz de ajustar automaticamente o DCO para um valor próximo de um múltiplo de uma freqüência de referência (o sinal ACLK proveniente do oscilador LFXT1).

Utilizando o FLL, a partir de um cristal de 32.768Hz, adequado à construção de um relógio de tempo real (RTC), é possível obter freqüências de *clock* muito mais altas e estáveis. Desta forma, uma aplicação pode manter um RTC funcionando, ao mesmo tempo em que a CPU

permanece num estado de baixa potência e quando for necessário, a CPU pode ser ativada e executar o programa em uma alta freqüência de *clock*. A estabilidade e precisão obtidas com o FLL dispensam um segundo oscilador a cristal.

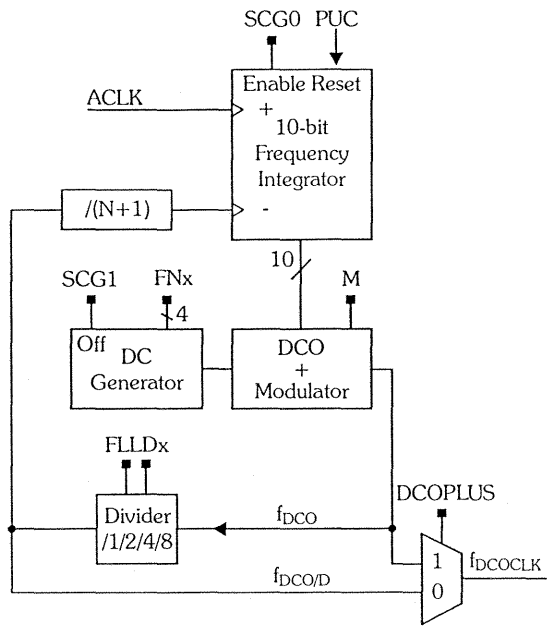


Figura 5-5

O sinal de saída do DCO é dividido por um fator "D" e posteriormente novamente dividido por um fator igual a "N+1". O sinal assim originado é aplicado na entrada de contagem decrescente de um contador *up/down* de 10 *bits* (integrador). Ao mesmo tempo, o sinal ACLK, proveniente do oscilador LFXT1 (que funciona como referência para o FLL), é aplicado na entrada de contagem crescente do mesmo contador. A saída desse contador realimenta o modulador do DCO, corrigindo e estabilizando a freqüência de operação.

Na prática, em poucas palavras, o integrador vai comparar os sinais de *clock* presentes nas suas entradas "+" e "-", gerando um sinal de erro que é utilizado para configurar o DCO para que a sua freqüência de saída seja aproximadamente igual a D*(N+1)*ACLK.

O fator "D" é selecionado entre 1, 2, 4 ou 8, de acordo com a configuração dos *bits* FLLDx (registrador SCFI0). O fator "N" é um valor de 7 *bits* carregado no registrador SCFGCTL (os seus 7 *bits* menos significativos).

O sinal de saída f_{DCOCLK} pode ser selecionado entre duas opções pelo *bit* DCOPLUS (registrador FLL_CTL0): f_{DCO} (quando DCOPLUS = 1) ou f_{DCO}/D (quando DCOPLUS = 0).

A fórmula básica para o cálculo da freqüência de oscilação do FLL é:

$$f_{DCO} = D*(N+1)*ACLK$$

É importante observar que nos *chips* da família 4xx, o funcionamento do DCO é ligeiramente diferente daquele encontrado nos *chips* da família 1xx. O DCO possui um ajuste de alcance de freqüência, selecionado pelos *bits* FN_8, FN_4, FN_3 e FN_2 (registrador SCFI0), permitindo que ele opere em diversas faixas de freqüência.

Repare que também é possível desativar o modulador pelo *bit* SCFG_M no registrador SCFGCTL. Com SCFQ_M = 0 o modulador está ativado, enquanto com SCFQ_M = 1 o modulador está desativado.

Além disso, devemos ressaltar que o DCO encontrado na série 4xx não foi projetado para ser utilizado autonomamente e sim para ser controlado e sincronizado pelo oscilador externo LFXT1.

Após um *reset*, o FLL é configurado para operar a aproximadament e 1MHz, quando utilizado um cristal de 32.768Hz no LFXT1.

5.3.5. Sinais de *Clock* Internos (MCLK, SMCLK e ACLK)

Como já foi dito, existem três sinais de *clock* internos na arquitetura MSP430: o *clock* principal (MCLK), o *clock* secundário (SMCLK) e o *clock* auxiliar (ACLK).

Os três sinais são gerados em diferentes fontes dentro do módulo oscilador do *chip*:

5.3.5.1. MCLK

Nas famílias 1xx e 2xx, a origem do sinal de *clock* principal pode ser selecionada entre três fontes diferentes: o oscilador LFXT1, o oscilador XT2 (se presente) e o DCO. A seleção de uma dessas fontes é feita pelos *bits* SELMx, localizados no registrador BSCCTL2. O sinal proveniente de uma dessas fontes pode ainda ser dividido por um fator de 1, 2, 4 ou 8, conforme os *bits* DIVMx, localizados no mesmo registrador. O diagrama da seleção do sinal de *clock* principal para as famílias 1xx e 2xx pode ser visto na figura 5-6.

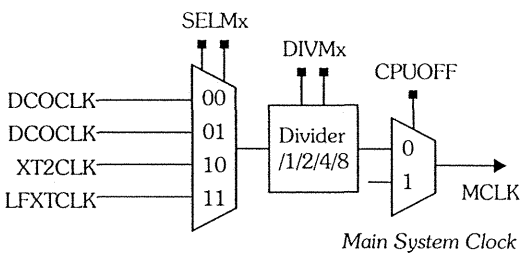


Figura 5-6

O sinal MCLK pode ser disponibilizado externamente, nos *chips* de 64 pinos, pelo pino P5.4, bastando configurar o pino para a função alternativa (registrador P5SEL).

No caso da família 4xx, a única diferença importante é a ausência do divisor após o MUX de seleção do sinal de origem do *clock*. Os *bits* de seleção da origem do *clock* principal SELMx estão localizados no registrador FLL_CTL1. Repare ainda que o sinal de *clock* auxiliar (ACLK) que aparece na figura 5-7 é o mesmo sinal LFXTCLK da figura 5-6. Na família 4xx, os dois sinais são idênticos, como veremos mais adiante.

O sinal MCLK pode ser disponibilizado externamente, nos *chips* da família 4xx, pelo pino P1.1, bastando configurar o pino para a função alternativa (registrador P1SEL).

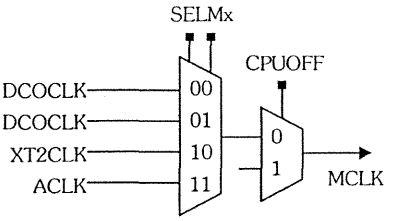


Figura 5-7

5.3.5.2. SMCLK

Nas famílias 1xx e 2xx, o sinal de *clock* secundário pode ter a sua origem selecionada entre duas alternativas: o DCO ou o oscilador XT2. A seleção é feita pelo *bit* SELS localizado no registrador BCSCTL2.

Nos *chips* que não possuem o oscilador XT2, a seleção dele provoca a utilização do oscilador LFXT1 como origem do sinal de *clock* secundário.

O sinal selecionado é em seguida dividido por um fator de 1, 2, 4 ou 8, selecionável pelos *bits* DIVSx, localizados no registrador BCSCTL2.

A figura 5-8 representa o diagrama típico de seleção do *clock* secundário para os *chips* das famílias 1xx e 2xx.

O sinal SMCLK pode ser disponibilizado externamente, pelo pino P1.4 (nos *chips* de 20 e 28 pinos) ou pelo pino P5.5 (*chips* de 64 pinos), bastando configurar o pino para a função alternativa (registrador P1SEL ou P5SEL).

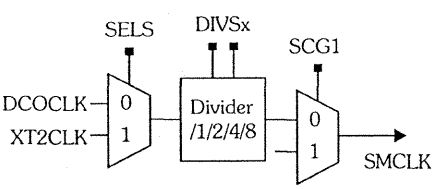


Figura 5-8

Em relação à família 4xx (figura 5-9), novamente são poucas as diferenças para as demais famílias. A principal é a ausência do divisor encontrado na figura 5-8.

Nos *chips* que não possuem o oscilador XT2, a única fonte de *clock* possível para o SMCLK é o sinal do DCO.

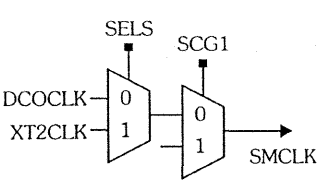


Figura 5-9

O sinal SMCLK pode ser disponibilizado externamente, nos *chips* da família 4xx com 80 ou 100 pinos, pelo pino P1.4, bastando configurá-lo para a função alternativa.

5.3.5.3. ACLK

Nas famílias 1xx e 2xx, o circuito de *clock* auxiliar conta basicamente com um divisor programável com fatores de 1, 2, 4 ou 8. Esses fatores são selecionados pelos *bits* DIVAx, localizados no registrador BCSCTL1. A figura 5-10 representa o diagrama desse circuito.

O sinal ACLK pode ser disponibilizado externamente, pelo pino P2.0 (nos *chips* com 20 ou 28 pinos) ou pelo pino P5.6 (nos *chips* com 64 pinos), bastando configurar o pino para a função alternativa (registrador P2SEL ou P5SEL).

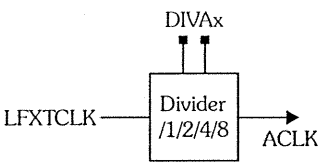


Figura 5-10

Na família 4xx, o sinal de *clock* auxiliar é exatamente o mesmo $f_{Crystal}$, que é o sinal de saída do oscilador LFXT1. Analisando o diagrama da figura 5-11, observamos que após o divisor programável temos um novo sinal chamado ACLK/n, que não está disponível para os periféricos internos. A seleção do fator de divisão é feita pelos *bits* FLL_DIVx, localizados no registrador FLL_CTL1.

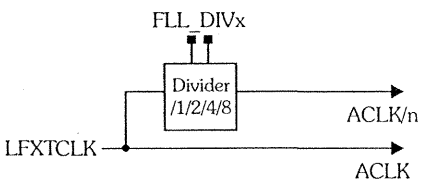


Figura 5-11

Nos *chips* da família 4xx, o sinal ACLK pode ser disponibilizado externamente por meio do pino P1.5, bastando configurá-lo para a função alternativa (registrador P1SEL).

5.3.6. Gerenciamento de Falha no Oscilador

O sistema de *clock* dos MSP430 inclui também um sistema de gerenciamento de falhas do oscilador.

Esse sistema é capaz de comutar automaticamente para o oscilador interno no caso de uma falha do oscilador externo. Ao mesmo tempo, também é possível gerar uma interrupção de forma a notificar o programa em execução dessa nova condição.

Família 1xx

Nessa família de *chips*, o sistema de gerenciamento de falhas do oscilador é um circuito que monitora o funcionamento do oscilador LFXT1 (no modo de alta freqüência) e do XT2.

Quando um dos osciladores, desde que ativo, deixa de oscilar por mais do que 50μs e o sinal de *clock* principal (MCLK) é derivado de um dos dois osciladores, ele é comutado automaticamente para trabalhar a partir do oscilador DCO.

Neste caso, o *flag* OFIFG (registrador IFG1) é setado, indicando a condição de falha do oscilador. Caso o *bit* OFIE (registrador IE1) esteja setado, uma interrupção NMI será disparada, provocando o desvio do programa para o endereço constante do vetor 14. A RTI deve providenciar a checagem do *flag* OFIFG de forma a diagnosticar a falha do oscilador.

O *flag* OFIFG permanece setado, mesmo que o oscilador causador da falha retorne ao seu funcionamento normal, cabendo ao programa do usuário a tarefa de apagá-lo. Uma vez apagado o *flag*, a fonte de *clock* retorna à sua seleção original.

Um detalhe importante a ser destacado é que quando se utiliza o oscilador LFXT1 operando no modo de baixa freqüência (LF), a fonte do *clock* principal (MCLK) pode ser alterada (entre o DCO e o LFXT1) independentemente do estado do *flag* OFIFG.

Neste caso, nos *chips* dotados do oscilador XT2, é possível apagar o *flag* OFIFG por *software*. Nos *chips* que não dispõem do XT2, não é possível apagar o *flag*. Isso, no entanto, não afeta a capacidade de alternar entre as fontes de *clock* principal.

Família 2xx

A família 2xx inclui um sistema de gerenciamento de falhas do oscilador mais avançado que o da família 1xx.

Existem dois *flags* responsáveis pela sinalização das seguintes condições de falha (considera-se uma condição de falha quando o oscilador é ativado e não gera um sinal de *clock* estável para o sistema):

- **LFXT1OF** - *falha* no oscilador LFXT1 quando opera tanto no modo de baixa freqüência quanto no modo de alta freqüência;
- **XT2OF** - falha no oscilador XT2.

Os *flags* LFXT1OF e XT2OF estão localizados no registrador BCSCTL3.

Qualquer um dos *flags*, uma vez setado, faz com que o *flag* de falha do oscilador OFIFG seja também setado.

Com a condição de falha do oscilador solucionada, o respectivo *flag* (LFXT1OF ou XT2OF) também é apagado, no entanto o *flag* OFIFG permanece setado, indicando a condição de falha.

Assim como na família 1xx, uma vez setado o OFIFG, a fonte de *clock* principal é automaticamente comutada para o DCO e permanece assim enquanto o *flag* estiver setado.

Família 4xx

Nos *chips* da família 4xx, o sistema de gerenciamento de falhas do oscilador é um pouco mais elaborado que nas outras famílias.

A principal diferença é que neste caso existem *flags* separados para determinar o oscilador originador da falha. Além disso, o gerenciamento de falhas é estendido também ao LFXT1 operando no modo de baixa freqüência (LF), opção não disponível na família 1xx.

Existem ao todo quatro *flags* de falha do oscilador (LFOF, XT1OF, XT2OF e DCOF), todos localizados no registrador FLL_CTL0:

- **LFOF** - ativado quando é detectada uma falha do oscilador LFXT1 (desde que ativado), quando opera no modo de baixa freqüência. O circuito FLL, caso ativo, conta regressivamente, na tentativa de travar na freqüência do oscilador, até que seja

selecionada a menor freqüência de operação do DCO. Neste caso, o *flag* DCOF também será ativado. O retorno da operação normal do LFXT1 apaga automaticamente os *flags* LFOF e algum tempo após, o FLL trava na freqüência do oscilador, fazendo com que o DCOF seja também apagado.

- **XT1OF** - ativado quando é detectada uma falha do oscilador LFXT1 operando no modo de alta freqüência (HF), desde que o oscilador esteja ativado.
- **XT2OF** - ativado quando é detectada uma falha do oscilador XT2, desde que ativado.
- **DCOF** - ativado no caso da seleção de uma freqüência muito baixa (*bits* DCOx = 0 no registrador SCFI1) ou muito alta (*bits* DCOx = 1 no registrador SCFI1) para o DCO.

Em todos os casos, a ativação do *bit* de falha do oscilador provoca a ativação do *flag* OFIFG (registrador IFG1). Uma vez setado, esse *flag* faz com que a fonte do *clock* principal seja comutada para o DCO. Caso o *bit* OFIE (registrador IE1) esteja setado, será gerada uma interrupção não-mascarável, fazendo com que o fluxo do programa desvie para o endereço apontado pelo vetor de interrupção número 14.

***Atenção:** Em todos os MSP430, o flag OFIFG (registrador IFG1) é sempre setado após um reset POR e uma vez setado, a fonte de clock principal passa a ser sempre o DCO. Para selecionar outra fonte de clock, é necessário apagar esse flag.*

5.3.7. Seleção das Fontes de *Clock*

A seleção da fonte de *clock* utilizada para os sinais MCLK e SMCLK é feita pelos *bits* SELM (**SEL**eção do **M**CLK) e e SELS (**SEL**eção de **S**MCLK). Com a fonte de *clock* alterada, ocorre um período de sincronização entre a fonte anterior e a selecionada. Esse período é normalmente de um ou dois ciclos de *clock* da fonte previamente ativa. Após a sincronização, o *clock* passa a ser fornecido pela nova fonte selecionada.

Nas tabelas seguintes, podemos observar a seleção das fontes do MCLK e do SMCLK:

SELM	Fonte do MCLK
00	oscilador DCO
01	oscilador DCO
10	oscilador XT2
11	oscilador LFXT1

Tabela 5-5

SELS	Fonte do SMCLK
0	oscilador DCO
1	oscilador XT2 (se presente); caso contrário, utiliza o oscilador LFXT1

Tabela 5-6

A seleção de uma ou outra fonte de *clock* é determinada basicamente pelo perfil da aplicação em execução, pelos periféricos em uso e pelas necessidades de consumo de energia. Podemos, por exemplo, configurar alguns periféricos para operar a partir do *clock* secundário (SMCLK) e com isso, utilizando os modos LPM0 ou LPM1, é possível reduzir o consumo da aplicação, mantendo somente os periféricos essenciais em operação e desativando a CPU e o MCLK.

Vale lembrar que o programa deve sempre verificar o estado do *flag* OFIFG (registrador IFG1) e apagá-lo caso ele esteja setado, pois nesse caso, a troca de fonte de *clock* não tem efeito (o oscilador permanece operando no modo de falha).

Relativamente ao sinal de *clock* auxiliar (ACLK), resta dizer que ele é sempre derivado do oscilador LFXT1, sem outra opção como fonte para ele.

5.3.8. Registradores do Módulo Oscilador

Os registradores utilizados ou associados ao módulo oscilador variam de acordo com a família de microcontroladores.

No caso da família 1xx, temos os seguintes registradores:

- **DCOCTL** - responsável pelo controle e seleção de freqüências do DCO;
- **BCSCTL1** - um dos registradores de controle do módulo oscilador (BCS);
- **BCSCTL2** - segundo registrador de controle do módulo oscilador;
- **BCSCTL3** - terceiro registrador de controle do módulo oscilador (somente disponível nos *chips* da família 2xx);
- **PxSEL** - seleção da função alternativa para os pinos do módulo oscilador.

5.3.8.1. DCOCTL

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0056	DCOCTL	Leitura / Escrita	DCOx			MODx				
		Reset	0	1	1	0	0	0	0	0

DCOx - seleção de freqüência do DCO (veja a tabela 5-2 na página 119) (símbolos **DCO2**, **DCO1** e **DCO0**);

MODx - seleção do modulador. Esses *bits* selecionam quantas vezes a freqüência f_{DCO+1} é utilizada num período de 32 ciclos do DCOCLK. Nos demais ciclos (32-MOD) é utilizada a freqüência f_{DCO} (símbolos **MOD4, MOD3, MOD2, MOD1** e **MOD0**).

5.3.8.2. BCSCTL1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0057	BCSCTL1	Leitura / Escrita	XT2OFF	XTS	DIVAx		XT5V	RSELx		
		Reset	1	0	0	0	0	1	0	0

XT2OFF - liga/desliga o oscilador XT2 (caso presente):
0 - oscilador XT2 ligado;
1 - oscilador XT2 desligado (símbolo **XT2OFF**).

XTS - seleção de modo do oscilador LFXT1:
0 - modo de baixa freqüência (LF);
1 - modo de alta freqüência (HF) (símbolo **XTS**).

DIVAx - divisor do sinal ACLK:

DIVAx	Fator de divisão	Símbolo
00	1	**DIVA_0**
01	2	**DIVA_1**
10	4	**DIVA_2**
11	8	**DIVA_3**

XT5V - não utilizado, manter sempre em zero;

RSELx - seleção do resistor interno (freqüências aproximadas válidas para a família 1xx) (símbolos **RSEL2, RSEL1** e **RSEL0**):

RSELx	DCOx							
	0	1	2	3	4	5	6	7
0	88,8KHz	97,3KHz	106,4KHz	116,9KHz	128,7KHz	141,9KHz	156,5KHz	176,8KHz
1	140,9KHz	154,5KHz	168,9KHz	185,4KHz	203,8KHz	224,6KHz	247,3KHz	279,7KHz
2	221,2KHz	242,3KHz	264,8KHz	290,9KHz	319,7KHz	352,3KHz	387,5KHz	439,1KHz
3	370,3KHz	405,3KHz	442,9KHz	486,5KHz	534,8KHz	589,8KHz	649,5KHz	736,9KHz
4	625,7KHz	685,1KHz	748,9KHz	823KHz	904,7KHz	998,9KHz	1,1MHz	1,25MHz
5	1,04MHz	1,13MHz	1,24MHz	1,36MHz	1,49MHz	1,66MHz	1,83MHz	2,08MHz
6	1,64MHz	1,79MHz	1,96MHz	2,16MHz	2,38MHz	2,64MHz	2,92MHz	3,31MHz
7	2,56MHz	2,79MHz	3,06MHz	3,37MHz	3,72MHz	4,13MHz	4,59MHz	5,19MHz

Nos microcontroladores da família 2xx, o registrador BCSCTL1 possui uma configuração ligeiramente diferente, em virtude da existência de 16 seleções diferentes para o resistor interno do oscilador:

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0057	BCSCTL1	Leitura / Escrita	XT2OFF	XTS	DIVAx		RSELx			
		Reset	1	0	0	0	0	1	1	1

5.3.8.3. BCSCTL2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0058	BCSCTL2	Leitura / Escrita	SELMx		DIVMx		SELS	DIVSx		DCOR
		Reset	0	0	0	0	0	0	0	0

SELMx - seleção da origem do *clock* principal (MCLK):

00 - DCO (símbolo **SELM_0**);

01 - DCO (símbolo **SELM_1**);

10 - oscilador XT2 ou oscilador LFXT1 (caso o XT2 não esteja presente) (símbolo **SELM_2**);

11 - oscilador LFXT1 (símbolo **SELM_3**).

DIVMx - fator de divisão do *clock* principal (MCLK):

DIVMx	Fator de divisão	Símbolo
00	1	**DIVM_0**
01	2	**DIVM_1**
10	4	**DIVM_2**
11	8	**DIVM_3**

SELS - seleção da origem do *clock* secundário (SMCLK):
0 - DCO;
1 - oscilador XT2 (quando presente) ou LFXT1 (símbolo **SELS**).

DIVSx - fator de divisão para o *clock* secundário (SMCLK):

DIVSx	Fator de divisão	Símbolo
00	1	**DIVS_0**
01	2	**DIVS_1**
10	4	**DIVS_2**
11	8	**DIVS_3**

DCOR - seleção do resistor do DCO:
0 - resistor interno;
1 - resistor externo (pino R_{OSC}) (símbolo **DCOR**).

5.3.8.4. BCSCTL3 (somente na família 2xx)

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0053	BCSCTL3	Leitura / Escrita	XT2Sx		LFXT1Sx		XCAPx		XT2OF	LFXT1OF
		Reset	0	0	0	0	0	1	0	1

XT2Sx - seleção da faixa de operação do oscilador XT2:
00 - cristal ou ressonador entre 0,4 e 1MHz;
01 - cristal ou ressonador entre 1 e 3MHz;
10 - cristal ou ressonador entre 3 e 16MHz;
11 - *clock* digital externo entre 0,4 e 16MHz.

LFXT1Sx - seleção da faixa de operação do oscilador LFXT1:
Quando o *bit* XTS=0:
00 - cristal de 32768Hz;
01 - reservado;
10 - reservado;
11 - *clock* digital externo.
Quando o *bit* XTS=1
00 - cristal ou ressonador entre 0,4 e 1MHz;
01 - cristal ou ressonador entre 1 e 3MHz;
10 - cristal ou ressonador entre 3 e 16MHz;
11 - *clock* digital externo entre 0,4 e 16MHz.

XCAPxPF - seleção dos capacitores de carga internos para o oscilador LFXT1:
00 - aproximadamente 1pF;
01 - aproximadamente 6pF;
10 - aproximadamente 10pF;
11 - aproximadamente 12,5pF.

XT2OF - indicador de falha do oscilador XT2:
0 - oscilador XT2 operando normalmente;
1 - oscilador XT2 em estado de falha.

LFXT1OF - indicador de falha do oscilador LFXT1:
0 - oscilador LFXT1 operando normalmente;
1 - oscilador LFXT1 em estado de falha.

Na família 4xx, temos os seguintes registradores responsáveis pelo controle do funcionamento do FLL+:

SCFGCTL - registrador de controle de *clock* do sistema;

SCFI0 - registrador 0 do integrador de freqüência;

SCFI1 - registrador 1 do integrador de freqüência;

FLL_CTL0 - registrador 0 de controle do FLL+;

FLL_CTL1 - registrador 1 de controle do FLL+;

PxSEL - seleção da função alternativa para os pinos do módulo oscilador.

5.3.8.5. SCFGCTL

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0052	SCFGCTL	Leitura / Escrita	SCFQ_M	N						
		Reset	0	0	0	1	1	1	1	1

SCFQ_M - controle de modulação:
0 - modulação do FLL ligada;
1 - modulação do FLL desligada (símbolo **SCFQ_M**);

N - fator de multiplicação da freqüência do DCO:
0x01 - símbolo **SCFQ_64K** (2 * ACLK);
0x03 - símbolo **SCFQ_128K** (4 * ACLK);
0x07 - símbolo **SCFQ_256K** (8 * ACLK);
0x0F - símbolo **SCFQ_512K** (16 * ACLK);
0x1F - símbolo **SCFQ_1M** (32 * ACLK);
0x3F - símbolo **SCFQ_2M** (64 * ACLK);
0x7F - símbolo **SCFQ_4M** (128 * ACLK).

5.3.8.6. SCFI0

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0050	SCFI0	Leitura / Escrita	FLLDx		FN_x				MODx(LSBs)	
		Reset	0	1	0	0	0	0	0	0

FLLDx - divisor do *loop* de realimentação do FLL+:

FLLDx	Fator de divisão	Símbolo
00	1	**FLLD_1**
01	2	**FLLD_2**
10	4	**FLLD_4**
11	8	**FLLD_8**

FN_x - seleção da faixa de freqüência de operação do DCO (símbolos **FN_2, FN_3, FN_4** e **FN_8**):

0000 - 650KHz a 6,1MHz;

0001 - 1,3MHz a 12.1MHz;

001x - 2MHz a 17.9MHz;

01xx - 2.8MHz a 26.6MHz;

1xxx - 4.2MHz a 46MHz.

MODx - *bits* menos significativos do modulador. Esses *bits* são modificados automaticamente pelo FLL+.

5.3.8.7. SCFI1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0051	SCFI1	Leitura / Escrita	DCOx					MODx (MSBs)		
		Reset	0	0	0	0	0	0	0	0

DCOx - seleção de faixa do DCO. Esses *bits* são modificados automaticamente pelo FLL+;

MODx - *bits* mais significativos do modulador são alterados automaticamente pelo FLL+.

5.3.8.8. FLL_CTL0

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0053	FLL_CTL0	Leitura / Escrita	DCOPLUS	XTS_FLL	XCAPxPF		XT2OF	XT1OF	LFOF	DCOF
		Reset	0	0	0	0	0	0	1	1

DCOPLUS - ativação do pré-divisor da saída do DCO. Esse pré-divisor permite multiplicar a freqüência de saída do DCO por um fator de 1, 2, 4 ou 8, conforme a seleção feita nos *bits* FLLDx (registrador SCFI0):

0 - divisor ativo;

1 - divisor inativo (símbolo **DCOPLUS**).

XTS_FLL - seleção do modo de operação do oscilador LFXT1:

0 - modo de baixa freqüência (LF);

1 - modo de alta freqüência (HF) (símbolo **XTS_FLL**).

XCAPxPF - seleção dos capacitores de carga internos para o oscilador LFXT1:

00 - aproximadamente 1pF (símbolo **OSCCAP_0**);

01 - aproximadamente 6pF (símbolo **OSCCAP_1**);

10 - aproximadamente 8pF (símbolo **OSCCAP_2**);

11 - aproximadamente 10pF (símbolo **OSCCAP_3**).

XT2OF - indicador de falha do oscilador XT2:

0 - oscilador XT2 operando normalmente;

1 - oscilador XT2 em estado de falha (símbolo **XT2OF**).

XT1OF - indicador de falha do oscilador LFXT1 no modo HF:

0 - oscilador operando normalmente;

1 - oscilador em estado de falha (símbolo **XT1OF**).

LFOF - indicador de falha do oscilador LFXT1 no modo LF:

0 - oscilador operando normalmente;

1 - oscilador em estado de falha (símbolo **LFOF**).

DCOF - indicador de falha do DCO:

0 - DCO operando normalmente;

1 - DCO em estado de falha (símbolo **DCOF**).

5.3.8.9. FLL_CTL1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0053	FLL_CTL1	Leitura / Escrita	não utilizado	SMCLK OFF	XT2OFF	SELMx		SELS	FLL_DIVx	
		Reset	0	0	1	0	0	0	0	0

SMCLKOFF -liga/desliga o sinal de *clock* secundário (não disponível nos modelos 41x e 42x):

0 - SMCLK ligado;

1 - SMCLK desligado (símbolo **SMCLKOFF**).

XT2OFF - liga/desliga o oscilador XT2 (não disponível nos modelos 41x e 42x):

0 - XT2 ligado;

1 - XT2 desligado (desde que não utilizado como fonte do MCLK ou do SMCLK) (símbolo **XT2OFF**).

SELMx - seleção da fonte do *clock* principal (MCLK) (não disponível nos modelos 41x e 42x, sendo o MCLK sempre derivado do DCO):

00 - DCO (símbolo **SELM_DCO**);

01 - DCO (símbolo **SELM_DCO**);

10 - oscilador XT2 (símbolo **SELM_XT2**);

11 - oscilador LFXT1 (símbolo **SELM_A**).

SELS - seleção da fonte do *clock* secundário (SMCLK) (não disponível nos modelos 41x e 42x, sendo o SMCLK sempre derivado do DCO):

0 - DCO;

1 - oscilador XT2 (símbolo **SELS**).

FLL_DIVx - seleção do divisor para o sinal ACLK (que origina o sinal ACLK/n):

FLL_DIVx	Fator de divisão	Símbolo
00	1	**FLL_DIV_0**
01	2	**FLL_DIV_1**
10	4	**FLL_DIV_2**
11	8	**FLL_DIV_3**

Comuns a todas as famílias, temos os seguintes registradores:

IE1 - registrador de controle de interrupções, no qual está localizado o *bit* OFIE;

IFG1 - registrador de sinalização de interrupções, no qual está localizado o *flag* OFIFG.

5.3.8.10. IE1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0000	IE1	Leitura / Escrita	UTXIE0	URXIE0	ACCVIE	NMIIE	-	-	**OFIE**	WDTIE
		Reset	0	0	0	0	0	0	0	0

OFIE - habilitação da interrupção por falha no oscilador (símbolo **OFIE**).

5.3.8.11. IFG1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0002	IFG1	Leitura / Escrita	UTXIFG0	URXIFG0	-	NMIIFG	RSTIFG	PORIFG	**OFIFG**	WDTIFG
		Reset	1	0	0	0	0	0	0	0

OFIFG - sinalizador de interrupção por falha no oscilador (símbolo **OFIFG**).

5.3.9. Exemplos de Configuração

5.3.9.1. BCS

Em seguida temos alguns exemplos de configuração do módulo de *clock* básico (BCS):

1. *Clock* principal (MCLK) e *clock* secundário (SMCLK) de aproximadamente 8MHz, operando a partir do DCO (com resistor externo). O *clock* auxiliar (ACLK) não estará operante, uma vez que o oscilador LFXT1 estará inoperante:

   Neste caso, o *chip* está operando apenas com *clock* interno. Para utilizarmos um resistor externo como fonte de corrente para o gerador DC do DCO, precisamos setar o

bit DCOR no registrador BCSCTL2. Utilizando um resistor de 150KOhms, um fator de divisão DCOCTL:DCOx = 7 e BCSCTL1:RSELx = 7, obtemos uma freqüência de aproximadamente 8,08MHz (veja a tabela 5-4).

Para operação nestas condições, é suficiente configurar os registradores do BCS como segue:

```
DCOCTL = 0xE0;
BCSCTL1 = 0x87;
BCSCTL2 = 0x01;
```

Ou, utilizando os símbolos definidos nos cabeçalhos ("io430xxx.h"):

```
DCOCTL = DCO2+DCO1+DCO0+MOD4;
BCSCTL1 = XT2OFF+RSEL2+RSEL1+RSEL0;
BCSCTL2 = DCOR;
```

Em *Assembly* poderíamos escrever:

```
MOV.B #0xE0, &DCOCTL
MOV.B #0x87, &BCSCTL1
MOV.B #0x01, &BCSCTL2
```

Observe que poderíamos obter uma freqüência ainda mais próxima de 8MHz se utilizássemos a facilidade do modulador. Com DCOCTL:DCOx = 6 e DCOCTL:MODx = 30 podemos obter uma freqüência de *clock* de aproximadamente 8,01MHz.

Para menor consumo de energia, poderíamos também desligar o oscilador LFXT1 (SR:OSCOFF = 1), já que ele não está sendo utilizado nessa configuração.

2. *Clock* principal (MCLK) de aproximadamente 4MHz operando a partir do DCO, sinal de *clock* secundário (SMCLK) operando na mesma freqüência do *clock* principal e *clock* auxiliar (ACLK) operando a partir do oscilador LFXT1, com um cristal de 32768Hz:

   Neste caso, temos de configurar o MCLK e o SMCLK para operar a partir do DCO e o ACLK para operar a partir do LFXT1, que será configurado para o modo de baixa freqüência (LF).

   A configuração do DCO para operar a aproximadamente 4MHZ é feita configurando o BCSCTL1:RSELx = 7 e DCOCTL:DCOx = 5 (tabela 5-3). Neste caso, temos uma freqüência de *clock* aproximada de 4,13MHz. Para maior precisão poderíamos selecionar o DCOx = 4 e por meio do modulador (MODx = 27) obtermos uma freqüência de aproximadamente 4,004 MHz.

   A configuração dos registradores do BCS deve ser feita como segue:

```
DCOCTL = 0x9B;
BCSCTL1 = 0x87;
BCSCTL2 = 0x00;
```

   Ou, utilizando os símbolos definidos nos cabeçalhos ("io430xxx.h"):

```
DCOCTL = DCO2+MOD4+MOD3+MOD2;
BCSCTL1 = XT2OFF+RSEL2+RSEL1+RSEL0;
BCSCTL2 = 0x00;
```

   Em *Assembly* poderíamos escrever:

```
MOV.B #0xE0, &DCOCTL
MOV.B #0x87, &BCSCTL1
CLR &BCSCTL2
```

3. *Clock* principal (MCLK) operando a partir do oscilador XT2 com um cristal de 8MHz, *clock* secundário (SMCLK) operando a partir do DCO (1MHz) e o *clock* auxiliar (ACLK) operando a 32768 Hz:

   Esse tipo de configuração demonstra a grande flexibilidade do sistema de *clock* dos MSP430. Podemos ter três fontes de *clock* distintas e independentes, com o *clock* auxiliar utilizado para um relógio de tempo real. A CPU poderia operar a partir do MCLK de 8MHz nos picos de processamento e nos momentos de baixa demanda ou de economia de energia, o MCLK poderia ser comutado para o DCO, desligando o XT2 e reduzindo o consumo de energia.

   A fonte de *clock* do MCLK será o oscilador XT2 (BCSCTL2:SELMx = 2), que deve estar ativado (BCSCTL1:XT2OFF = 0). O *clock* secundário será derivado do DCO (BCSCTL2:SELS = 0), operando a 1MHz e o *clock* auxiliar operando a partir do oscilador LFXT1 a 32768Hz.

   A configuração do DCO para operar a aproximadamente 1MHZ é feita configurando o BCSCTL1:RSELx = 4, DCOCTL:DCOx = 5 (tabela 5-3). Neste caso, temos uma freqüência de *clock* aproximada de 998 KHz.

   A configuração dos registradores do BCS deve ser feita como segue:

```
DCOCTL = 0xA0;
BCSCTL1 = 0x04;
BCSCTL2 = 0x80;
```

   Ou, utilizando os símbolos definidos nos cabeçalhos ("io430xxx.h"):

```
DCOCTL = DCO2+DCO0;
BCSCTL1 = RSEL2;
BCSCTL2 = SELM_2;
```

   Em *Assembly* poderíamos escrever:

```
MOV.B #0xA0, &DCOCTL
MOV.B #0x04, &BCSCTL1
MOV.B #0x80, &BCSCTL2
BIC.B #OFIFG, &IFG1
```

   Lembre-se de que o MCLK somente passará a ser fornecido pelo XT2 após o apagamento do *flag* OFIFG (registrador IFG1).

5.3.9.2. FLL+

A seguir, temos exemplos de configuração dos registradores para a operação nas principais modalidades de funcionamento do FLL+:

1. *Clock* principal (MCLK) de 32.768Hz a partir do LFXT1:

   Neste caso, utilizamos um cristal de 32.768Hz ligado aos pinos XIN e XOUT do oscilador LFXT1. O sinal de *clock* secundário (SMCLK) será originado do DCO com uma freqüência de aproximadamente 4MHz (corrigida pelo oscilador LFXT1). O sinal de *clock* auxiliar (ACLK) originado do oscilador LFXT1 será igual a 32.768Hz, assim como o *clock* principal (MCLK).

O fator de multiplicação do FLL deverá ser igual a:

$$\frac{F_{DESEJADA}}{ACLK} = \frac{4MHz}{32768Hz} = 122{,}07$$

Carregando o registrador SCFQCTL com o valor 0x79 (121 decimal), teremos um fator de multiplicação do FLL igual a:

$$1*(N+1) = 1*(121+1) = 122$$

Esta configuração resulta em uma freqüência de operação igual a 32768Hz*122 = 3,997MHz.

Para configurar o FLL+ para operar como descrito anteriormente, precisamos modificar os seguintes registradores:

```
SCFQCTL = 0x79;
SCFI0 = 0x06;
FLL_CTL0 = 0;
FLL_CTL1 = 0x38;
// aguarda o OFIFG ser apagado
while (!IFG1_bit.OFIFG) IFG1_bit.OFIFG = 0;
```

Ou utilizando os símbolos definidos nos cabeçalhos ("io430xxx.h"):

```
SCFQCTL = 0x79;
SCFI0 = FN_2;
FLL_CTL0 = 0;
FLL_CTL1 = SELM_A + XT2OFF;
// aguarda o OFIFG ser apagado
while (!IFG1_bit.OFIFG) IFG1_bit.OFIFG = 0;
```

Em *Assembly* poderíamos escrever:

```
MOV.B #0x79,&SCFQCTL
MOV.B #0x06,&SCFI0
CLR.B &FLL_CTL0
MOV.B #0x38,&FLL_CTL1
BIC.B #OFIFG, &IFG1
```

2. *Clock* principal (MCLK) de 8MHz sincronizado por um cristal de 32.768Hz:

Neste caso, utilizamos um cristal de 32.768Hz ligado aos pinos XIN e XOUT do LFXT1. Esse oscilador fornece o sinal ACLK na mesma freqüência do cristal. Esse sinal pode ser utilizado para sincronização de periféricos como os *timers* e/ou o controlador de LCD. O *clock* principal da CPU é gerado pelo FLL, corrigido e estabilizado pelo LFXT1. Neste exemplo, utilizaremos uma freqüência de 8MHz. O sinal SMCLK também operará nessa freqüência, sendo gerado pelo FLL.

O fator de multiplicação do FLL deverá ser igual a:

$$\frac{F_{DESEJADA}}{ACLK} = \frac{8MHz}{32768Hz} = 244{,}14$$

Como o maior valor para "N" é igual a 127 (0x7F), será necessário utilizar um fator de multiplicação igual a 2. Desta forma, o valor de "N" será também multiplicado por 2.

Carregando o registrador SCFQCTL com o valor 0x79 (121 decimal), teremos um fator de multiplicação do FLL igual a:

$$2*(N+1)=2*(121+1)=244$$

Esta configuração resulta em uma freqüência de operação igual a 32.768Hz*244 = 7,995MHz.

Para operação nesta modalidade os seguintes registradores devem ser alterados:

```
SCFQCTL = 0x79;
SCFI0 = 0x46;
FLL_CTL0 = 0x80;
FLL_CTL1 = 0;
```

Ou utilizando os símbolos definidos nos cabeçalhos ("io430xxx.h"):

```
SCFQCTL = 0x79;
SCFI0 = FLLD_2 + FN_2;
FLL_CTL0 = DCOPLUS;
FLL_CTL1 = 0;
// aguarda o OFIFG ser apagado
while (!IFG1_bit.OFIFG) IFG1_bit.OFIFG = 0;
```

A operação descrita pode ser realizada em *Assembly* com as seguintes instruções:

```
MOV.B #0x79,&SCFQCTL
MOV.B #0x48,&SCFI0
MOV.B #0x80,&FLL_CTL0
CLR.B &FLL_CTL1
BIC.B #OFIFG, &IFG1
```

3. *Clock* principal de 8MHz a partir do oscilador XT2:

   Neste caso, temos o módulo FLL+ operando a partir do oscilador XT2, o *clock* principal (MCLK) e o *clock* secundário (SMCLK) serão derivados do XT2, enquanto o *clock* auxiliar permanecerá inoperante (uma vez que o oscilador LFXT1 não esteja ativo).

   Para operação nesta modalidade, os seguintes registradores devem ser alterados:

```
FLL_CTL0 = 0x0;
FLL_CTL1 = SELS + SELM_XT2;
```

   A operação descrita pode ser realizada em *Assembly* com o programa seguinte:

```
CLR.B &FLL_CTL0
MOV.B #SELS + SELM_XT2,&FLL_CTL1
BIC.B #OFIFG, &IFG1
```

Atenção: *Em todos os MSP430, o flag OFIFG é sempre setado após um reset POR e uma vez setado, a fonte de clock principal passa a ser sempre o DCO. Para selecionar outra fonte de clock, é necessário apagar esse flag!*

5.4. Portas de E/S

Os MSP430 podem ter até seis portas de entrada/saída de uso geral, cada uma com até oito pinos, totalizando até 48 pinos de E/S. Cada um desses pinos pode ser configurado individualmente para funcionar como uma entrada ou saída. Adicionalmente, alguns pinos possuem outras funções multiplexadas, que podem ser ativadas ou desativadas pelo programa em execução.

A nomenclatura adotada pelo fabricante para as portas é P1, P2, P3, P4, P5 e P6. As portas P1 e P2 possuem ainda a funcionalidade de poderem gerar uma interrupção na transição dos sinais de entrada.

Cada porta possui um conjunto básico de registradores os quais controlam a sua operação: um registrador de entrada para a leitura do estado dos pinos da porta (PxIN), um registrador de saída para a escrita de valores nos pinos da porta (PxOUT), um registrador para o controle individual da direção dos pinos da porta, se entrada ou saída (registrador PxDIR), e um registrador para a seleção da função do pino, que seleciona entre a função de E/S normal e a função eventualmente multiplexada ao pino (registrador PxSEL). Um *bit* em nível "1" no PxSEL seleciona a função alternativa para o respectivo pino, enquanto um nível "0" seleciona a função normal.

Sendo um determinado pino configurado como saída (o respectivo *bit* do PxDIR = 1), o nível lógico presente nesse pino será aquele configurado pelo *bit* do registrador PxOUT.

Caso o pino seja configurado como entrada (o respectivo *bit* do PxDIR = 0), o nível lógico externo presente no pino pode ser lido pelo registrador PxIN.

No caso de o pino estar configurado como saída e se realizar a leitura do registrador PxIN, o valor lido será o do estado lógico do pino, que na maioria dos casos corresponde ao estado escrito no registrador PxOUT. Repare que, em alguns casos, a afirmação anterior pode não ser verdadeira. Tudo vai depender da carga ligada ao pino e do instante em que se fará a leitura.

As portas P1 e P2, além dos registradores anteriores, incluem registradores específicos para a função de interrupção: um registrador sinalizador de interrupção do pino (PxIFG), um registrador de seleção de borda de sensibilidade de cada pino (PxIES) e um registrador de habilitação individual de interrupção para cada pino da porta (PxIE).

A função de interrupção disponível nas portas P1 e P2 permite que uma transição de "0" para "1" ou de "1" para "0" em um dos pinos dessas portas provoque a interrupção do programa em execução, desde que o respectivo *bit* do registrador PxIE esteja setado, assim como o controle global de interrupções (GIE).

A rotina de tratamento da interrupção deve providenciar o apagamento do respectivo *flag* no registrador PxIFG, de forma a evitar uma nova interrupção causada pelo mesmo evento externo.

A duração mínima do pulso em um dos pinos das portas P1 ou P2 deve ser de 1,5 períodos do MCLK ou mais. Pulsos menores que este podem não ser reconhecidos pelo *hardware* do microcontrolador.

Nos *chips* da família 2xx, temos ainda disponíveis resistores internos de *pull-up* e *pull-down* que podem ser ativados/desativados pelo *software* do usuário. O registrador PxREN controla a ativação e a desativação dos resistores. Quando um resistor está ativado (*bit* do PxREN = 1), o registrador PxOUT passa a controlar se o resistor é de *pull-up* (*bit* do PxOUT = 1) ou *pull-down* (*bit* do PxOUT = 0).

A capacidade máxima de corrente de cada pino configurado como saída (tanto de fornecimento quanto de drenagem) é de 6mA, com um limite de 48mA para todo o conjunto de portas.

No caso de um pino configurado como entrada, a corrente drenada é da ordem de aproximadamente 50nA.

Os MSP430 dispõem ainda de circuitos *Schmitt Trigger* em todas as portas, bem como no pino RST/NMI e nos pinos da interface JTAG.

Como elemento de proteção, todos os pinos do *chip* possuem diodos de proteção (*clamping*) capazes de limitar a máxima tensão no pino em aproximadamente 0,3V acima da tensão de alimentação do *chip* (V_{DD}) e a mínima tensão no pino em aproximadamente 0,3V abaixo do terra de referência (V_{SS}) de alimentação do *chip*. A máxima corrente admitida em tais diodos é de aproximadamente 2mA.

A existência de tais diodos, além de oferecer proteção, facilita a interface do MSP430 com sistemas alimentados por 5V, pois com a simples inclusão de um resistor em série com o pino, o *chip* pode ler sinais provenientes de sistemas alimentados com 5V (quando alimentado com 3,3 Volts).

***Atenção:** Os pinos de E/S não utilizados devem, preferencialmente, ser configurados como saídas e deixados desconectados, de forma a reduzir o consumo e aumentar a imunidade a ruídos do circuito. Após um reset, os pinos são sempre configurados como entradas e com a função alternativa desativada.*

5.4.1. Registradores das Portas de E/S

Como já vimos, existem quatro registradores básicos de acesso às portas de E/S:

- **PxOUT** - para escrita nos pinos de uma determinada porta (P1OUT para a porta 1, P2OUT para a porta 2 e assim por diante);
- **PxIN** - para a leitura do estado dos pinos de uma determinada porta;
- **PxDIR** - para configurar a direção individual dos pinos de uma determinada porta (entrada ou saída);
- **PxSEL** - responsável pela seleção da função alternativa do pino (0 - função E/S normal, 1 - função alternativa).

As portas 1 e 2 dispõem ainda de funções adicionais de interrupção, controladas pelos seguintes registradores:

- **PxIFG** - responsável pela sinalização de uma mudança de estado em um dos pinos da porta;
- **PxIES** - utilizado para controlar a borda de sensibilidade à interrupção da porta;
- **PxIE** - responsável pela habilitação das interrupções nas portas.

Os *chips* da família 2xx dispõem ainda de outro registrador para o controle dos resistores de *pull-up/pull-down*:

- **PxREN** - utilizado para habilitação dos resistores de *pull-up/pull-down*.

```
#include <msp430x14x.h>
        NAME    main
        PUBLIC  main
        ORG     0FFFEh
        DC16    main
        RSEG    CODE
main    MOV       #0x3FF,SP    ; inicializa o apontador da pilha
        MOV       #WDTPW+WDTHOLD,WDTCTL ; desliga o watchdog
        BIS.B     #1,P1DIR     ; configura o pino P1.0 como saída
loop    BIT.B     #2,P1IN      ; testa o pino p1.1
        JZ        acende       ; desvia se P1.1 = 0
        BIC.B     #1,P1OUT     ; limpa a saída P1.0
        BR        #loop        ; desvia para o loop
acende  BIS.B     #1,P1OUT     ; seta a saída P1.0
        BR        #loop        ; desvia para o loop
        END       main
```

Exemplo 5-4

```
#include <io430x14x.h>

#define led P1OUT_bit.P1OUT_0

int main( void )
{
  WDTCTL = WDTPW + WDTHOLD; // desativa o watchdog
  P1DIR_bit.P1DIR_0 = 1;    // configura o pino P1.0 como saída
  while (1)
  {
    // lê o estado do pino P1.1
    // caso esteja setado, apaga o LED, caso esteja em zero, liga o LED
    if (P1IN_bit.P1IN_1) LED =0; else LED =1;
  }
}
```

Exemplo 5-5

2. Disponibilizando externamente os sinais de *clock* do sistema. Para permitir que um dos sinais internos de *clock* (MCLK, SMCLK ou ACLK) seja disponibilizado externamente, basta setar o *bit* PxSEL do respectivo pino.

   No caso de um *chip* da família 1xx, como o MSP430F149, poderíamos utilizar a seguinte instrução *Assembly* para ativar a saída do MCLK no pino P5.4:

   ```
   BIS.B #16, &P5SEL
   ```

   Ou ainda:

   ```
   BIS.B #BIT4, &P5SEL
   ```

   Em C poderíamos escrever:

   ```
   P5SEL |= 16; // OU bit a bit entre P5SEL e o valor 16 decimal
   ```

   Ou ainda:

   ```
   P5SEL_bit.P5SEL_4 = 1; // utilizando a biblioteca io430xxxx.h
   ```

5.5. Timer A

O *timer* A é um contador/temporizador de 16 *bits* com as seguintes características:

- Um contador assíncrono progressivo/regressivo de 16 *bits* com módulo programável e capacidade de interrupção.
- Capacidade de operar a partir de diversas fontes de *clock* interno e externo.
- Até três registradores de CCP - Captura/Comparação/PWM (cinco nos modelos F415, F417, FW423, FW425 e FW427), capazes de operar no modo de captura (medição de período de sinais), comparação (geração de pulsos de largura programável) e PWM (geração de sinais de freqüência e ciclo ativo programáveis).
- Os modelos F415, F417, FW423, FW425 e FW427 implementam dois *timer* A: um com três canais CCP (TA0) e outro com cinco canais CCP (TA1).
- Capacidade de captura originada a partir da saída do comparador analógico.
- Possibilidade de iniciar uma conversão A/D a partir da saída de comparação 1 (canal 1).

Na figura 5-13 temos o diagrama em blocos da estrutura do *timer* A para três canais de captura/comparação.

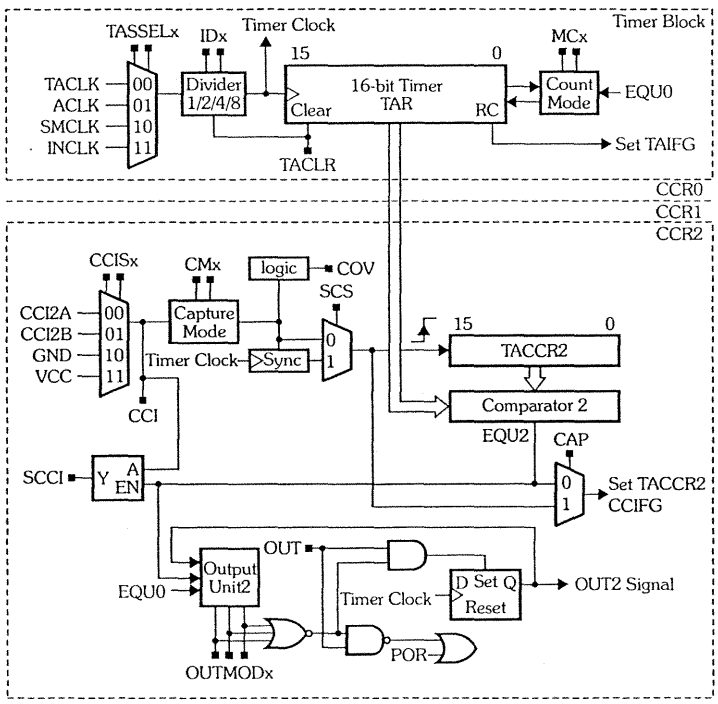


Figura 5-13

O princípio de funcionamento do *timer* é relativamente simples. O coração do sistema é o contador de 16 *bits* (TAR) que é alimentado por um sinal de *clock* selecionável de uma das quatro fontes possíveis (TACLK (*clock* externo), ACLK, SMCLK ou INCLK). A seleção é feita pelos *bits* TASSELx (registrador TACTL). O sinal pode ainda ser dividido por um fator programável (1, 2, 4 ou 8), o que pode ser selecionado pelos *bits* IDx (registrador TACTL).

Esse sinal de *clock* provoca o incremento ou decremento da contagem do contador TAR. A direção de contagem dele é selecionada pelos *bits* MCx (registrador TACTL). Note que os *bits* MCx controlam muito mais que apenas a direção de contagem do TAR. Na verdade, eles selecionam o seu modo de operação, conforme a tabela 5-7:

MCx	Modo
00	Contagem parada
01	Contagem de módulo (até atingir o módulo de contagem)
10	Contagem contínua (de 0 até 0xFFFF)
11	Contagem progressiva/regressiva

Tabela 5-7

No modo 0 (MCx = 00), a contagem do TAR permanece parada, mantendo o seu último valor. Nenhum incremento ou decremento acontece nesse modo.

No modo 1 (MCx = 01), o contador TAR passa a contar progressivamente a cada pulso de *clock* até atingir o valor programado no registrador TACCR0 (módulo de contagem). Ao atingir esse valor, o sinal EQU0 é ativado, provocando o reinício da contagem do TAR a partir do valor 0. Simultaneamente, o *flag* TAIFG é setado, indicando o transbordo da contagem do *timer*. Caso essa interrupção esteja habilitada, o programa será desviado para o devido vetor de interrupção.

Observe que, caso o programa do usuário altere o registrador TACCR0 para um valor menor que a contagem atual do TAR, a contagem dele retorna a zero (um pulso adicional de *clock* pode ser necessário antes do efetivo retorno a zero).

Para calcularmos o período da interrupção TAIFG neste modo, podemos utilizar a seguinte fórmula:

$$T_{INT} = \frac{1}{F_{CLK}/\text{Prescaler}/(\text{TACCR0}+1)}$$

sendo:

- T_{INT} - Período da interrupção TAIFG em segundos;
- F_{CLK} - Freqüência da fonte de *clock* em Hz;
- *Prescaler* - Fator de divisão do *prescaler* de entrada do *timer*;
- TACCR0 - Módulo da contagem armazenado no registrador TACCR0.

No modo 2 (MCx = 10), o contador TAR realiza a contagem progressiva desde 0x0000 até 0xFFFF, perfazendo um total de 65.536 contagens. Após atingir o valor 0xFFFF, no próximo pulso de *clock*, a contagem retorna a 0 e o *flag* TAIFG é setado. Caso essa interrupção esteja habilitada, o programa será desviado para o devido vetor de interrupção.

O período de interrupção nesse modo é calculado pela seguinte fórmula:

$$T_{INT} = \frac{1}{F_{CLK}/\text{Prescaler}/65536}$$

sendo:

- T_{INT} - Período da interrupção TAIFG em segundos;
- F_{CLK} - Freqüência da fonte de *clock* em Hz;
- *Prescaler* - Fator de divisão do *prescaler* de entrada do *timer*.

No modo 3 (MCx = 11), o contador TAR realiza a contagem progressiva desde 0 até o valor programado no registrador TACCR0. Ao atingir esse valor, a direção da contagem é invertida e o contador passa a contar regressivamente até 0. Ao atingir esse valor, a direção da contagem é novamente invertida e todo o processo tem início novamente.

O período de interrupção nesse modo é calculado pela seguinte fórmula:

$$T_{INT} = \frac{1}{F_{CLK}/\text{Prescaler}/(\text{TACCR0} * 2)}$$

sendo:

- T_{INT} - Período da interrupção TAIFG em segundos;
- F_{CLK} - Freqüência da fonte de *clock* em Hz;
- *Prescaler* - Fator de divisão do *prescaler* de entrada do *timer*;
- TACCR0 - Módulo da contagem armazenado no registrador TACCR0.

Caso o programa do usuário altere o conteúdo do registrador TACCR0 e a contagem seja decrescente, a contagem vai correr normalmente até atingir o valor zero. Neste caso, o novo período somente terá efeito na próxima contagem progressiva.

Já no caso de o contador estar no modo progressivo e o novo valor armazenado em TACCR0 ser maior que o antigo, o contador contará até esse novo valor. Caso o novo valor seja menor que o antigo, a contagem será invertida e o contador passará a contar regressivamente até zero. Observe que uma contagem adicional pode ocorrer antes que a direção da contagem seja efetivamente invertida.

5.5.1. *Reset* do Contador

O contador TAR pode ser *ressetado* de diversas formas:

1. Pela escrita do valor 0 no registrador TAR;
2. Pela escrita do valor 0 no registrador TACCR0 (desde que o contador não esteja no modo 2 (modo contínuo)). Além do *reset* do contador, essa opção também tem o efeito de paralisar a contagem, que permanecerá *ressetada* até que o conteúdo do registrador TACCR0 seja alterado para um valor diferente de zero;
3. Setando o *bit* TACLR (registrador TACTL) que possui a finalidade específica de *reset* do contador TAR. Uma vez setado, o conteúdo do registrador TAR é zerado, bem como o divisor de *clock* e a direção de contagem.

5.5.2. Modo de Captura

Uma das funcionalidades do *timer* A é o modo de captura, que pode ser utilizado para medição de período de sinais ou outras medições de tempo em que se requeiram máxima precisão e mínima intervenção da CPU.

O funcionamento do modo de captura é bastante simples. O sinal a ser medido é amostrado por uma das entradas CCIxA ou CCIxB (selecionáveis pelos *bits* CCISx no registrador TACCTLx). A borda de sensibilidade da captura (subida, descida ou ambas) é selecionada pelos *bits* CMx localizados no registrador TACCTLx.

Com a borda selecionada, a contagem atual do TAR é armazenada no registrador do TACCRx do respectivo canal e o *flag* CCIFG (registrador TACCTLx) é setado.

Observe que uma captura também pode ser iniciada por *software,* configurando os *bits* CM para 11 (captura em ambas as bordas), setando o *bit* CCIS1 e em seguida alternando o estado do *bit* CCIS0. A cada alteração do CCIS0 será realizada uma captura.

O período do sinal será igual à diferença entre duas medições consecutivas. Desta forma é possível a medição de até três (ou cinco) períodos simultâneos, utilizando os três (ou cinco) canais de captura/comparação do *timer* A.

Observe que a captura pode ocorrer tanto de forma assíncrona (ou seja, não sincronizada com o *clock* do contador) quanto de forma síncrona (sincronizada com o *clock* do contador). Essa opção é selecionada pelo *bit* SCS (registrador TACCTLx).

Como a captura na modalidade assíncrona pode introduzir erros inerentes à eventual diferença de temporização relativamente ao *clock* do contador, é preferível que se utilize a captura síncrona (SCS = 1).

O *timer* A disponibiliza ainda um controle de forma a sinalizar uma eventual sobrescrita do valor capturado, o que é feito pelo *bit* COV (registrador TACCTLx). A operação desse *bit* ocorre da seguinte forma: se após uma comparação o conteúdo do registrador TACCRx não for lido pelo *software*, caso ocorra uma nova comparação, o novo período será armazenado no TACCRx, ocasionando a perda do valor anterior. Neste caso, o *bit* COV é setado, indicando esta condição.

Uma facilidade adicional disponível nos *chips* dotados de comparador analógico é que uma captura pode ser disparada automaticamente pelo sinal CAOUT de saída do comparador, permitindo a construção de conversores A/D. Basta configurar a entrada de comparação do canal 1 para o comparador analógico (TACCTL1:CCIS = 1). Para maiores detalhes veja o tópico 5.13.

As conexões do bloco de captura do *timer* A podem ser vistas na tabela 5-10.

5.5.3. Modo de Comparação/PWM

Além do modo de captura, outra modalidade de funcionamento do *timer* A é o modo de comparação e PWM, que pode ser utilizado principalmente para geração de pulsos ou interrupções com intervalos de tempo precisos ou também para a geração de sinais PWM.

Esse modo é selecionado apagando o *bit* CAP (registrador TACCTLx). O seu funcionamento está relacionado diretamente às unidades de saída (*Output Units*) existentes no *hardware* do *timer*.

Basicamente, uma unidade de saída consiste em um comparador digital, que compara a contagem do TAR com o valor programado num dos registradores TACCRx. Quando as duas contagens são iguais, é gerado um sinal EQUx (em que x é o número do canal CCR).

Esse sinal EQUx é utilizado para gerar o sinal de comparação OUTx, conforme definido pelos *bits* OUTMODx (registrador TACCTLx) na tabela 5-8.

Os modos 3 e 7 são os mais indicados para a geração de sinais PWM. O registrador TACCR0 é utilizado para determinar o período do sinal e o registrador TACCRx determina o ciclo ativo. No modo 3 temos um sinal PWM ativo em nível 0 e no modo 7 temos um sinal PWM ativo em nível 1.

Modo	OUTMODx	Modo	Descrição
0	000	Saída direta	Neste modo, o estado do sinal OUTx é definido diretamente pelo estado do *bit* OUT (registrador TACCTLx).
1	001	Setar	A saída é setada quando a contagem do TAR atinge o valor programado no TACCRx. A saída permanece setada até a seleção de outro modo de saída ou um *reset* do *timer*.
2	010	Inverter/ apagar	A saída tem seu estado lógico invertido quando a contagem do TAR atinge o valor do TACCRx e apagada quando a contagem do TAR atinge o valor do TACCR0. Este modo não é útil no canal 0, tendo em vista que ele é utilizado para o controle do período do sinal.
3	011	Setar/ apagar	A saída é setada quando a contagem do TAR atinge o valor do TACCRx e apagada quando a contagem do TAR atinge o valor do TACCR0. Este modo não é útil no canal 0, tendo em vista que é utilizado para o controle do período do sinal.
4	100	Inverter	A saída tem seu estado lógico invertido cada vez que a contagem do TAR atinge o valor programado no TACCRx. O período do sinal de saída é igual ao dobro do período de contagem total do *timer*.
5	101	Apagar	A saída é apagada quando a contagem do TAR atinge o valor programado no TACCRx. A saída permanece apagada até a seleção de outro modo de saída ou um *reset* do *timer*.
6	110	Inverter/ setar	A saída tem o seu estado lógico invertido cada vez que a contagem do TAR atinge o valor programado no TACCRx e setada quando a contagem do TAR atinge o valor programado no TACCR0. Este modo não é útil no canal 0, tendo em vista que ele é utilizado para o controle do período do sinal.
7	111	Apagar/ setar	A saída é apagada quando a contagem do TAR atinge o valor programado no TACCRx e setada quando a contagem do TAR atinge o valor programado no TACCR0. Este modo não é útil no canal 0, tendo em vista que é utilizado para o controle do período do sinal.

Tabela 5-8

Na figura 5-14 podemos observar um gráfico demonstrando o funcionamento dos diversos modos de comparação com o *timer* A operando no modo de contagem de módulo.

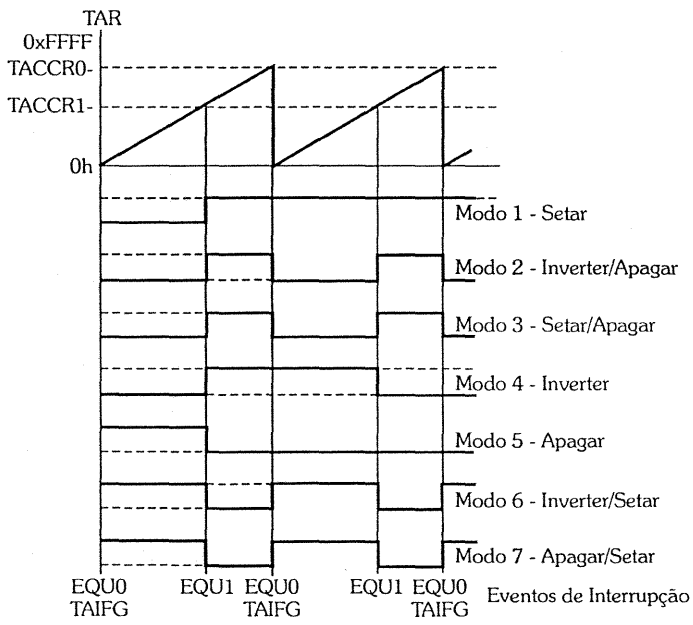


Figura 5-14

Os modos 2 e 6 são indicados para a geração de sinais PWM complementares e separados por uma banda morta (*deadband*) quando ambos os sinais permanecem inativos. A finalidade dessa banda morta é, por exemplo, impedir que dois elementos complementares possam conduzir simultaneamente em uma ponte H, o que acarretaria destruição de um ou mais elementos de comutação.

Na figura 5-15 podemos observar uma representação do funcionamento dos modos 2 e 6 gerando um sinal PWM com a existência de uma banda morta. O sinal de saída é obtido nos canais 1 e 2 (pois foi utilizado o canal 0 como controle de período da onda).

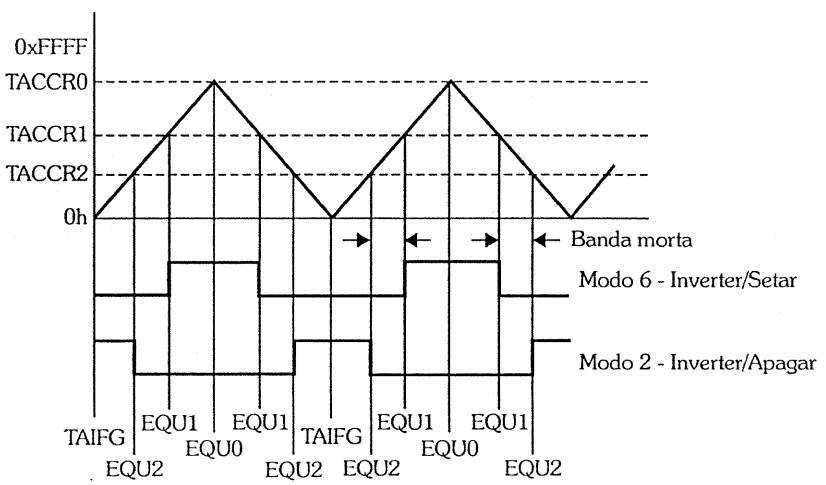


Figura 5-15

Pela análise do gráfico da figura 5-15 podemos facilmente concluir que o período da banda morta, no exemplo anterior, será igual a:

$$T_{DEAD} = T_{TIMER} * (TACCR1 - TACCR2)$$

Note que, como os registradores TACCRx não possuem *buffers* de escrita temporária, após a escrita de um valor, ele entra em vigor imediatamente. Sendo assim, o programador deve tomar providências durante a alteração do ciclo ativo dos PWMs, de forma a preservar a largura mínima necessária para a banda morta.

Outro detalhe importante é que na alteração do modo de funcionamento do módulo de comparação, o programador deve ter o cuidado de não apagar todos os *bits* OUTMODx, pois provocaria a imediata entrada em funcionamento do modo 0, o que pode provocar ruídos ou transições bruscas no sinal de saída. Uma boa prática neste caso é primeiramente selecionar o modo 7 e em seguida selecionar o modo desejado, apagando os *bits* necessários.

As conexões de saída do bloco de comparação/PWM do *timer* A podem ser vistas na tabela 5-10.

Observe que, opcionalmente, é possível configurar a saída de comparação do canal 1 (OUT1 ou TA1) para iniciar uma conversão do ADC12. Basta configurar os *bits* TACCTL1:CCIS = 3. Maiores detalhes sobre esta operação encontram-se no tópico 5.15.

Nos *chips* dotados de conversor A/D de 10 *bits* (F11xx e F12x), é possível utilizar qualquer um dos sinais de saída do *timer* A (OUT0 ou TA0, OUT1 ou TA1, ou ainda OUT2 ou TA2) para iniciar uma conversão do ADC10. Maiores detalhes estão no tópico 5.14.

5.5.4. Interrupções do *Timer* A

O *timer* A possui quatro fontes de interrupções distintas (seis nos *chips* com cinco canais CCP). Essas fontes de interrupção possuem dois vetores distintos: um para a interrupção do canal 0 (TACCR0:CCIFG) e outro para as demais interrupções (estouro da contagem do *timer* (TAIFG) e interrupções dos demais canais CCP).

Caso ocorra uma interrupção do canal 0 do *timer* e ela esteja habilitada (TACCTL0:CCIE = 1 e SR:GIE = 1), o fluxo do programa será desviado para o vetor indicado na tabela de vetores de interrupção do *chip* (veja a tabela 5-1 para maiores informações sobre o vetor, em particular para cada modelo de *chip*) e o *flag* TACCTL0:CCIFG será automaticamente apagado.

No caso das demais interrupções do *timer*, o processo é um pouco diferente. Como nesse caso temos diversas interrupções compartilhando o mesmo vetor, torna-se necessário um mecanismo que permita identificar qual delas provocou o desvio do programa.

Esse mecanismo existe sob a forma do gerador de vetor de interrupção (TAIV). Cada uma das fontes de interrupção que o compõem (TAIFG, TACCR1:CCIFG até TACCR4:CCIFG) possui uma prioridade diferente. A fonte com maior prioridade carrega um valor especial no registrador TAIV.

Esse valor pode ser utilizado pela RTI para determinar qual das diferentes fontes foi a origem da interrupção, seja pela análise do valor (já que cada fonte possui um valor diferente), seja pela adição do valor ao conteúdo do PC, provocando um desvio calculado do programa. **Interrupções que não estejam habilitadas não modificam o registrador TAIV !**

Uma outra característica interessante desse mecanismo de interrupção é que a simples leitura ou escrita do TAIV automaticamente limpa o *flag* de interrupção que provocou o desvio do programa. No entanto, se outros *flags* do *timer* estiverem ativos, o programa será novamente desviado para a RTI e o processo se repete para o evento que estava pendente.

5.5.5. Conexões do *Timer* A

As tabelas seguintes apresentam as conexões de *clock*, entrada e saída, do *timer* A (o pino ou sinal é válido para todos os modelos de *chip*, a não ser que especificado em contrário):

Seleção TASSEL			
00 - TACLK	01 - ACLK	10 - SMCLK	11 - INCLK/ $\overline{\text{TACLK}}$
P1.0 (1xx) P1.5 (4xx) P1.5 (TA0) (*2) P2.5 (TA1) (*2)	ACLK (*1)	SMCLK (*1)	P2.1 (1xx) P1.5 (4xx) P1.5 (TA0) (*2) P2.5 (TA1) (*2)

(*1) - sinais internos ao *chip*;

(*2) - modelos 415, 417 e FW42x, com dois *timers* A (3+5 canais CCP).

Tabela 5-9

Sinal	Seleção CCIS			
	00 - CCIxA	01 - CCIxB	10 - GND	11 - V_{CC}
CCI0	P1.1 P1.0 (4xx) P1.4 (*2)	P2.2 P1.1 (4xx) P1.3 (*2)	GND	V_{CC}
CCI1	P1.2 P2.1 (*2)	CAOUT(*1)	GND	V_{CC}
CCI2	P1.3 P2.0 (4xx) P2.2 (*2)	ACLK(*1) SIFO0 (*2)	GND	V_{CC}
CCI3	P2.3 (*2)	SIFO1 (*2)	GND	V_{CC}
CCI4	P2.4 (*2)	SIFO2 (*2)	GND	V_{CC}
OUT0	P1.1 P1.0 (4xx) P1.4 (*2)	P1.5	P2.7	-
OUT1	P2.3 (11xx, 12xx) P1.2 P2.1 (*2)	P1.2 (11xx, 12xx) P1.6	P1.6 (11xx, 12xx) P2.3	ADC12 (*1)
OUT2	P2.4 (11xx, 12xx) P1.3 P2.0 (4xx) P2.2 (*2)	P1.3 (11xx, 12xx) P1.7	P1.7 (11xx, 12xx) P2.4	-
OUT3	P2.3 (*2)	-	-	-
OUT4	P2.4 (*2)	-	-	-

(*1) - sinais internos ao *chip*;

(*2) - modelos 415, 417 e FW42x.

Tabela 5-10

5.5.6. Registradores do *Timer* A

Os registradores utilizados no *timer* A estão listados em seguida:

- **TACTL** - registrador de controle do *timer* A;
- **TA1CTL** - registrador de controle do *timer* A nos *chips* com 5 canais CCP;
- **TAR** - registrador de acesso à contagem do *timer* A;
- **TA1R** - registrador de acesso à contagem do *timer* A nos *chips* com 5 canais CCP;
- **TACCTLx** - registradores de controle dos canais de captura/comparação/ PWM (TACCTL0, TACCTL1 e TACCTL2);
- **TA1CCTLx** - registradores de controle dos canais de captura/comparação/ PWM nos *chips* com 5 canais CCP (TA1CCTL0, TA1CCTL1, TA1CCTL2, TA1CCTL3 e TA1CCTL4);
- **TACCRx** - registradores de captura/comparação (TACCR0, TACCR1 e TACCR2);
- **TA1CCRx** - registradores de captura/comparação nos *chips* com 5 canais CCP (TA1CCR0, TA1CCR1, TA1CCR2, TA1CCR3 e TA1CCR4);
- **TAIV** - registrador sinalizador de vetor de interrupção do *timer* A;
- **TA1IV** - registrador sinalizador de vetor de interrupção do *timer* A nos *chips* com 5 canais CCP;
- **PxSEL** - seleção da função alternativa para os pinos do *timer* A.

5.5.6.1. TACTL

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0160 0x0180	TACTL TA1CTL	Leitura Escrita	Não utilizados						TASSELx	
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura Escrita	IDx		MCx		Não utilizado	TACLR	TAIE	TAIFG
		Reset	0	0	0	0	0	0	0	0

TASSELx - seleção da fonte de *clock* do *timer* A:
00 - TACLK (*clock* externo) (símbolo **TASSEL_0**);
01 - *clock* auxiliar (ACLK) (símbolo **TASSEL_1**);
10 - *clock* secundário (SMCLK) (símbolo **TASSEL_2**);
11 - INCLK (nos *chips* 4xx é o sinal TACLK invertido) (símbolo **TASSEL_3**).

IDx - divisor do *clock* do *timer* A:

IDx	Fator de divisão	Símbolo
00	1	**ID_0**
01	2	**ID_1**
10	4	**ID_2**
11	8	**ID_3**

MCx - seleção do modo de funcionamento do *timer* A:
00 - parado (símbolo **MC_0**);
01 - contagem progressiva com módulo (0 até o valor de TACCR0) **(símbolo MC_1)**;
10 - contínuo (0 até 65.535 e depois reinicia do zero) **(símbolo MC_2)**;
11 - crescente/decrescente (contagem de 0 até TACCR0 e depois até 0 novamente) **(símbolo MC_3)**.

TACLR - *reset* do *timer* A: setando esse *bit*, a contagem do TAR, do divisor de *clock* e a direção de contagem são apagadas. O *bit* retorna automaticamente a zero após o *reset* do *timer* (símbolo **TACLR**);

TAIE - habilitação da interrupção de estouro de contagem do TAR:
0 - interrupção desabilitada;
1 - interrupção habilitada (símbolo **TAIE**).

TAIFG - sinalizador de interrupção do TAR:
0 - nenhuma interrupção pendente;
1 - o TAR estourou a contagem e há uma interrupção pendente (símbolo **TAIFG**).

5.5.6.2. TAR

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
		Leitura	TARx							
		Escrita								
		Reset	0	0	0	0	0	0	0	0
0x0170	TAR		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0190	TA1R	Leitura	TARx							
		Escrita								
		Reset	0	0	0	0	0	0	0	0

TARx - 16 *bits* do registrador de contagem do *timer* A.

5.5.6.3. TACCTLx

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0162	TACCTL0	Leitura	CMx		CCISx		SCS	SCCI	Não utilizado	CAP
0x0164	TACCTL1	Escrita						-		
0x0166	TACCTL2	*Reset*	0	0	0	0	0	0	0	0
0x0182	TA1CCTL0		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0184	TA1CCTL1	Leitura	OUTMODx			CCIE	CCI	OUT	COV	CCIFG
0x0186	TA1CCTL2	Escrita					-			
0x0188	TA1CCTL3									
0x018A	TA1CCTL4	*Reset*	0	0	0	0	0	0	0	0

CMx - seleção do modo de captura:
00 - não realiza capturas (símbolo **CM_0**);
01 - captura na borda de subida (símbolo **CM_1**);
10 - captura na borda de descida (símbolo **CM_2**);
11 - captura em ambas as bordas (símbolo **CM_3**).

CCISx - seleção da entrada de captura/comparação:
00 - CCIxA (símbolo **CCIS_0**);
01 - CCIxB (símbolo **CCIS_1**);
10 - GND (símbolo **CCIS_2**);
11 - V_{CC} (símbolo **CCIS_3**).

SCS - sincronização da entrada de captura com o *clock* interno:
0 - captura assíncrona;
1 - captura síncrona (símbolo **SCS**).

SCCI - entrada sincronizada de captura/comparação. Esse sinal é sincronizado com o sinal EQUx e permite ler o estado da entrada CCI (símbolo **SCCI**).

CAP - seleção do modo de captura/comparação:
0 - modo de comparação;
1 - modo de captura (símbolo **CAP**).

OUTMODx - seleção do modo de comparação:

000 - a saída recebe o valor do *bit* OUT (símbolo **OUTMOD_0**);

001 - a saída é setada na comparação (símbolo **OUTMOD_1**);

010 - a saída é invertida na comparação e apagada ao atingir o valor do TACCR0 (símbolo **OUTMOD_2**);

011 - a saída é setada na comparação e apagada ao atingir o valor do TACCR0 (símbolo **OUTMOD_3**);

100 - a saída é invertida a cada comparação (símbolo **OUTMOD_4**);

101 - a saída é apagada na comparação (símbolo **OUTMOD_5**);

110 - a saída é invertida na comparação e setada ao atingir o valor do TACCR0 (símbolo **OUTMOD_6**);

111 - a saída é apagada na comparação e setada ao atingir o valor do TACCR0 (símbolo **OUTMOD_7**).

CCIE - habilitação da interrupção de captura/comparação do canal:

0 - interrupção desabilitada;

1 - interrupção habilitada (símbolo **CCIE**).

CCI - entrada de captura/comparação: permite ler o estado do sinal da entrada de captura/comparação (símbolo **CCI**).

OUT - estado do sinal de saída (OUTx) do canal. No modo 0, esse *bit* controla diretamente o estado da saída de comparação:

0 - saída em nível lógico "0";

1 - saída em nível lógico "1" (símbolo **OUT**).

COV - indicador de *overflow* de captura, ou seja, uma segunda captura foi feita antes da leitura da primeira:

0 - não ocorreu *overflow* de captura;

1 - ocorreu um *overflow* de captura (símbolo **COV**).

CCIFG - sinalizador de interrupção de captura/comparação:

0 - nenhuma interrupção pendente;

1 - uma interrupção de captura/comparação está pendente (símbolo **CCIFG**).

5.5.6.4. TACCRx

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0172	TACCR0	Leitura	TACCRx							
0x0174	TACCR1	Escrita								
0x0176	TACCR2	*Reset*	0	0	0	0	0	0	0	0
0x0192	TA1CCR0		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0194	TA1CCR1									
0x0196	TA1CCR2	Leitura	TACCRx							
0x0198	TA1CCR3	Escrita								
0x019A	TA1CCR4	*Reset*	0	0	0	0	0	0	0	0

TACCRx - 16 *bits* do registrador de captura/comparação do canal x. Lembrando que nos modos 1 e 3 do *timer*, o registrador TACCR0 funciona como registrador de módulo de contagem do TAR.

5.5.6.5. TAIV

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x012E 0x011E	TAIV TA1IV	Leitura	0	0	0	0	0	0	0	0
		Escrita	-	-	-	-	-	-	-	-
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	0	0	0	0	TAIVx			0
		Escrita	-	-	-	-				-
		Reset	0	0	0	0	0	0	0	0

TAIV	Fonte de interrupção	*Flag* de interrupção	Prioridade
0x00	Nenhuma interrupção	-	
0x02	CCP canal 1	TACCR1:CCIFG TA1CCR1:CCIFG *	Maior
0x04	CCP canal 2	TACCR2:CCIFG TA1CCR2:CCIFG *	
0x06	CCP canal 3 *	TA1CCR3:CCIFG *	
0x08	CCP canal 4 *	TA1CCR4:CCIFG *	
0x0A	Estouro do *timer*	TACTL:TAIFG	
0x0C	Reservado	-	
0x0E	Reservado	-	Menor

* nos *chips* com cinco canais CCP

Tabela 5-11

5.5.7. Exemplos de Utilização

Neste primeiro exemplo, vamos demonstrar como gerar uma onda quadrada com período configurável utilizando o modo de comparação do *timer* A.

O princípio de funcionamento é simples. O *timer* A está configurado para utilizar como fonte de *clock* o sinal SMCLK (aproximadamente 800KHz quando o *chip* sai do *reset*), assim, após passar pelo divisor de entrada (programado para dividir o sinal por oito), o sinal de *clock* que chega ao TAR é de aproximadamente 100KHz, o que implica que cada unidade de contagem do TAR vale aproximadamente 10μs.

Utilizando o canal CCP 1, no modo de comparação número quatro (inversão da saída a cada comparação), podemos gerar um sinal quadrado com período configurável, utilizando um pequeno artifício: a cada comparação o valor equivalente à metade do período é somado ao conteúdo do registrador de comparação (TACCR1), desta forma a cada intervalo programado ocorre uma nova comparação e a saída tem o seu estado lógico invertido.

Com o valor de inicialização de 25.000 carregado na variável intervalo (o que resulta em um semiciclo de aproximadamente 25.000*10μs=250ms), um LED conectado à saída P1.2 deve piscar a uma freqüência aproximada de 2Hz. Esse tipo de abordagem permite obter praticamente qualquer período para o sinal gerado, bastando respeitar os tempos de processamento da CPU (no que diz respeito à latência de interrupção e tempo de processamento da RTI).

```
#include <io430x14x.h>
#include <intrinsics.h>
unsigned int intervalo;
#pragma vector = TIMERA1_VECTOR
__interrupt void trata_timer(void)
{
  switch (TAIV)
  {
    case 0x02 : // interrupção do canal 1
                TACCR1 += intervalo;
                break;
    case 0x04 : // interrupção do canal 2
                break;
    case 0x06 : // interrupção do canal 3
                break;
    case 0x08 : // interrupção do canal 4
                break;
    case 0x0A : // interrupção de estouro do timer
                break;
  }
}

int main( void )
{
  WDTCTL = WDTPW + WDTHOLD;      // desativa o watchdog
  P1DIR_bit.P1DIR_2 = 1; // configura o pino P1.2 como saída
  // configura o timer A:
  // TASSEL_2 -> fonte de clock = SMCLK
  // ID_3 -> divisão por 8 do clock de entrada
  // MC_2 -> modo contínuo (contagem de 0 a 65535)
  TACTL = TASSEL_2+ID_3+MC_2;
  intervalo = 25000;
  TACCR1 = intervalo;            // ponto de disparo do CCP1
  // configura o canal CCP1:
  // OUTMOD_4 -> modo de saída 4 (inversão da saída a cada comparação)
  // CCIE -> interrupção de comparação do CCP1 habilitada
  TACCTL1 = OUTMOD_4+CCIE;
  P1SEL_bit.P1SEL_2 = 1;         // função alternativa para P1.2
  __bis_SR_register(GIE);        // habilita as interrupções
  while(1);                      // loop infinito
}
```

Exemplo 5-6

No próximo exemplo, temos uma demonstração da configuração do canal CCP1 para o modo de PWM. Considerando o *clock* interno de aproximadamente 800KHz, a freqüência do sinal PWM será igual a 800KHz / (TACCR0+1) = 800KHz / 16.385 = 48,83 Hz. Como estamos utilizando o modo de saída 7, a saída é apagada após cada comparação e ativada ao atingir o fim do período do sinal (determinado pelo registrador TACCR0). O tempo que a saída permanece ativa é definido pelo conteúdo do registrador TACCR1. A atualização do ciclo ativo é realizada na RTI que trata a interrupção do canal CCP0.

Um LED conectado ao pino P1.2 (saída TA1 nos *chips* F1xx, como o F149 utilizado neste exemplo) deve variar a intensidade do seu brilho, conforme o valor do ciclo ativo setado pela variável "tempo_ativo".

```
#include <io430x14x.h>
#include <intrinsics.h>
unsigned int tempo_ativo;

#pragma vector = TIMERA0_VECTOR
__interrupt void trata_ccp0(void)
{
  TACCR1 = tempo_ativo;
}

int main( void )
{
  WDTCTL = WDTPW + WDTHOLD;      // desativa o watchdog
  P1DIR_bit.P1DIR_2 = 1; // configura o pino P1.2 como saída
  P2DIR_bit.P2DIR_4 = 1;
  // configura o timer A:
  // TASSEL_2 -> fonte de clock = SMCLK
  // MC_1 -> modo de contagem de módulo (0 até TACCR0)
  TACTL = TASSEL_2+MC_1;
  TACCR0 = 16384;                // período = 16384 contagens do TAR
  tempo_ativo = 4096;            // ciclo ativo de 25%
  TACCR1 = tempo_ativo;
  // configura o canal CCP1:
  // OUTMOD_7 -> modo de saída 7 (apaga a saída na comparação,
  // seta quando atinge o TACCR0 )
  TACCTL1 = OUTMOD_7;
  // configura o canal CCP0:
  // CCIE -> interrupção de comparação do CCP1 habilitada
  TACCTL0 = CCIE;
  P1SEL_bit.P1SEL_2 = 1; // função alternativa para P1.2
  __enable_interrupt();  // habilita as interrupções
  while(1);              // loop infinito
}
```

Exemplo 5-7

5.6. *Timer B*

O *timer* B pode ser considerado uma evolução do *timer* A e o seu diagrama em blocos resumido pode ser visto na figura 5-16.

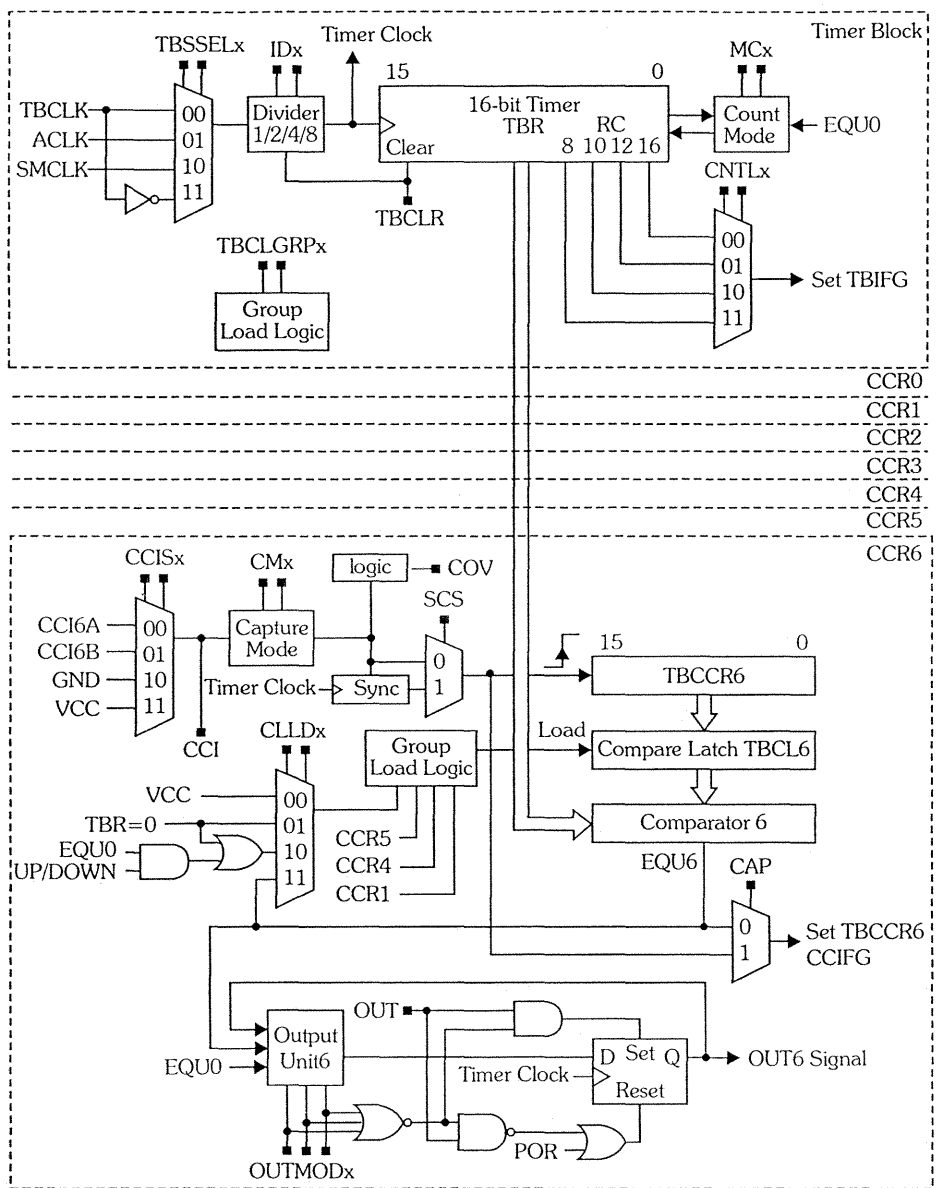


Figura 5-16

O *timer* B inclui praticamente todas as funcionalidades disponíveis no *timer* A e ainda algumas novidades:

- Capacidade de configurar o contador principal para quatro larguras diferentes: 16, 12, 10 ou 8 *bits*;
- Três ou sete canais de Captura/Comparação/PWM (CCP), dependendo do modelo do *chip* utilizado (os modelos F13x, F15x, F43x e FG43x possuem três canais, enquanto os modelos 14x, 16x e 44x possuem sete canais);
- *Latches* para o armazenamento temporário dos registradores de comparação;
- Agrupamento dos canais;
- Capacidade de colocar as saídas do *timer* em estado de alta impedância por meio de um sinal de controle externo;
- Capacidade de iniciar uma conversão do ADC após uma comparação no canal 0 ou 1.

Não vamos discutir todo o funcionamento do *timer*, pois ele é praticamente idêntico ao *timer* A.

Vamos nos concentrar apenas nas suas particularidades, iniciando pela largura de contagem programável.

5.6.1. Largura Programável

A largura do contador TBR (equivalente ao contador TAR do *timer* A) pode ser programada pelo usuário. O *timer* B pode funcionar como um contador de 8, 10, 12 ou 16 *bits*, o que pode ser selecionado pelos *bits* CNTLx (registrador TBCTL).

O programador deve estar ciente de que, alterando a contagem máxima do TBR, implicações ocorrem no funcionamento do modo de contagem de módulo e no modo de contagem progressiva/regressiva.

Quando for utilizado o modo de contagem de módulo (TBCTL:MC = 1), o programador deve tomar o cuidado de não programar o módulo de contagem (registrador TBCCR0) para um valor superior à contagem máxima do TBR, pois neste caso, o módulo de contagem obviamente nunca será atingido e o *timer* funcionará como se estivesse no modo contínuo.

No caso do modo de contagem progressiva/regressiva (TBCTL:MC = 3), se o valor do módulo de contagem for superior à contagem máxima do TBR, o *timer* não pode inverter o seu sentido de contagem e funciona como se estivesse programado para o modo contínuo, sem fazer contagens regressivas.

5.6.2. *Latches* de Comparação

Uma das maiores inovações do *timer* B diz respeito à adição dos *latches* TBCLx à arquitetura do módulo de comparação/PWM.

Como vimos no estudo do *timer* A, a geração de sinais no modo de comparação/PWM pode causar ruídos durante a atualização dos registradores de comparação, pois o conteúdo dos registradores TACCRx é utilizado diretamente para efetuar a comparação com o conteúdo do TAR. Tudo depende do momento em que o registrador TACCRx é alterado.

A solução para isso foi a adição de registradores intermediários (TBCLx) cujo conteúdo é utilizado diretamente para efetuar a comparação com o registrador TBR. Esses registradores não são diretamente acessíveis ao usuário, mas o seu conteúdo é atualizado pelo conteúdo dos registradores TBCCRx.

O instante em que ocorre a atualização de um registrador TBCLx é selecionado pelos *bits* CLLDx (registrador TBCCTLx):

- **Atualização imediata** - nesse modo, o conteúdo do registrador TBCLx é imediatamente atualizado no momento da escrita no TBCCRx.
- **Atualização no retorno a zero** - o conteúdo do TBCLx é atualizado no momento em que a contagem do TBR atinge o valor 0.
- **Atualização no retorno a zero ou ao atingir o módulo** - o conteúdo do TBCLx é atualizado no momento em que a contagem do TBR atinge o valor 0, ou quando a contagem do TBR atinge o módulo da contagem (que é ditado pelo valor armazenado no TBCL0).
- **Atualização na comparação** - o conteúdo do registrador TBCLx é atualizado após uma comparação bem-sucedida (ou seja, no instante em que o TBR atinge o valor previamente armazenado no TBCLx).

5.6.3. Agrupamento de Canais

Outra facilidade adicional do *timer* B é a possibilidade de agrupar canais, de forma que cada grupo tenha os seus registradores de comparação atualizados simultaneamente. Essa facilidade permite ao programador gerar sinais PWM com banda morta programável e sem os problemas de manutenção existentes no *timer* A. Como os canais são atualizados simultaneamente, o *dead time* (ou banda morta) é mantido sempre constante (desde que a aplicação faça essa previsão).

O agrupamento de canais é controlado pelos *bits* TBCLGRPx (registrador TBCTL), sendo possível criar os seguintes grupos:

TBCLGRPx	Canais	Controle de atualização
01	CCP1 + CCP2 CCP3 + CCP4 CCP5 + CCP6	CCP1 CCP3 CCP5
10	CCP1 + CCP2 + CCP3 CCP4 + CCP5 + CCP6	CCP1 CCP4
11	CCP0 + CCP1 + CCP2 + CCP3 + CCP4 + CCP5+ CCP 6	CCP1

Tabela 5-12

Quando formamos um grupo de canais, a atualização dos registradores TBCLx pertencentes ao grupo é controlada pelo registrador de controle de atualização representado na tabela 5-12. A configuração dos *bits* CLLDx desse registrador determina o instante em que os registradores serão atualizados. Repare que o modo de atualização não deve ser o imediato (CLLDx = 0), já que nesse modo a atualização é imediata e neste caso, o agrupamento perde a função. A configuração de atualização dos demais canais pertencentes ao grupo é ignorada, enquanto o agrupamento perdurar.

A atualização do grupo de registradores TBCLx pressupõe duas condições importantes:

1. Todos os registradores TBCCRx do grupo precisam estar atualizados (mesmo que o seu valor não seja alterado).
2. O evento de atualização configurado pelos *bits* CLLDx do canal de controle precisa ocorrer.

Sem a satisfação de ambas não ocorre atualização do grupo de canais.

5.6.4. Saídas Configuráveis para Modo de Alta Impedância

Outra característica adicional presente no *timer* B é o sinal TBOUTH. Trata-se de um sinal de origem externa que se estiver ativado (nível lógico "1"), coloca todas as saídas do *timer* B em modo de alta impedância.

Esse tipo de facilidade é normalmente utilizado em aplicações de controle, como medida de proteção de caráter emergencial, ou seja, detectada uma condição crítica pelo *hardware* externo (etapa de potência), essa linha permite desativar as saídas do *timer* sem nenhuma intervenção do *software*. Essa facilidade garante maior segurança e um baixo tempo de atraso na resposta a falhas.

O sinal TBOUTH está normalmente multiplexado no pino P5.7 (*chips* da família 1xx) ou no pino P1.3 (*chips* da família 4xx).

5.6.5. Disparo de Conversão do ADC após uma Comparação

Assim como no *timer* A, o *timer* B também possui a capacidade de iniciar automaticamente uma conversão A/D a partir dos canais 0 (CCP0) ou 1 (CCP1).

O sinal de saída de comparação do canal (OUT0 ou TB0, OUT1 ou TB1) pode ser direcionado internamente para provocar o início de uma conversão do ADC12.

Basta configurar os *bits* CCISx para o valor 1 (TBCCTLx:CCIS = 1). Maiores detalhes sobre este procedimento encomtram-se no tópico 5.15.

5.6.6. Interrupções do *Timer* B

O *timer* B possui quatro fontes de interrupções distintas (oito nos *chips* com sete canais CCP). Essas fontes de interrupção possuem dois vetores distintos: um para a interrupção do canal 0 (TBCCR0:CCIFG) e outro para a demais interrupções (estouro da contagem do *timer* (TBIFG) e interrupções dos demais canais CCP).

Se ocorrer uma interrupção do canal 0 do *timer* e ela estiver habilitada (TBCCTL0:CCIE =1 e SR:GIE = 1), o fluxo do programa será desviado para o vetor indicado na tabela de vetores de interrupção do *chip* (veja a tabela 5-1 para maiores informações sobre o vetor, em particular para cada modelo de *chip*) e o *flag* TBCCTL0:CCIFG será automaticamente apagado.

As demais interrupções utilizam o mesmo tipo de mecanismo gerador de vetor de interrupção descrito para o *timer* A. Neste caso, temos sete valores diferentes para o registrador TBIV, conforme podemos observar no tópico 5.6.8.5. As mesmas observações e recomendações feitas para o gerador de vetores de interrupção do *timer* A valem para o do *timer* B.

5.6.7. Conexões do *Timer* B

As tabelas seguintes apresentam as conexões de *clock*, entrada e saída, do *timer* B (o pino ou sinal é válido para todos os modelos de *chip*, a não ser que especificado em contrário):

Seleção TASSEL			
00 - TBCLK	01 - ACLK	10 - SMCLK	11 - INCLK/ $\overline{\text{TBCLK}}$
P4.7 (1xx) P1.4 (4xx)	ACLK (*1)	SMCLK (*1)	P4.7 (1xx) P1.4 (4xx)

(*1) - sinais internos ao *chip*.

Tabela 5-13

Sinal	Seleção CCIS			
	00 - CCIxA	01 - CCIxB	10 - GND	11 - V_{CC}
CCI0	P4.0 (1xx) P2.1 (4xx)	P4.0 (1xx) P2.1 (4xx)	GND	V_{CC}
CCI1	P4.1 (1xx) P2.2 (4xx)	P4.1 (1xx) P2.2 (4xx)	GND	V_{CC}
CCI2	P4.2 (1xx) P2.3 (4xx)	P4.2 (1xx) P2.3 (4xx)	GND	V_{CC}
CCI3	P4.3 (1xx) P3.4 (4xx)	P4.3 (1xx) P3.4 (4xx)	GND	V_{CC}
CCI4	P4.4 (1xx) P3.5 (4xx)	P4.4 (1xx) P3.5 (4xx)	GND	V_{CC}
CCI5	P4.5 (1xx) P3.6 (4xx)	P4.5 (1xx) P3.6 (4xx)	GND	V_{CC}
CCI6	P4.6 (1xx) P3.7 (4xx)	ACLK (*1)	GND	V_{CC}
OUT0	P4.0 (1xx) P2.1 (4xx)	ADC12 (*1)	-	-
OUT1	P4.1 (1xx) P2.2 (4xx)	ADC12 (*1)	-	-
OUT2	P4.2 (1xx) P2.3 (4xx)	-	-	-
OUT3	P4.3 (1xx) P3.4 (4xx)	-	-	-
OUT4	P4.4 (1xx) P3.5 (4xx)	-	-	-
OUT5	P4.5 (1xx) P3.6 (4xx)	-	-	-
OUT6	P4.6 (1xx) P3.7 (4xx)	-	-	-

(*1) - sinais internos ao *chip*.

Tabela 5-14

5.6.8. Registradores do *Timer* B

Os registradores utilizados no *timer* B estão listados em seguida:

- **TBCTL** - registrador de controle do *timer* B;
- **TBR** - registrador de acesso à contagem do *timer* B;
- **TBCCTLx** - registradores de controle dos canais de captura/comparação/PWM (TBCCTL0 a TBCCTL4 (nos *chips* com cinco canais CCP) ou até TBCCTL6 (nos *chips* com sete canais CCP));
- **TBCCRx** - registradores de captura/comparação (TBCCR0 a TBCCR4 (nos *chips* com cinco canais CCP) ou até TBCCR6 (nos *chips* com sete canais CCP));
- **TBIV** - registrador sinalizador de vetor de interrupção do *timer* B;
- **PxSEL** - seleção da função alternativa para os pinos do *timer* B.

5.6.8.1. TBCTL

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0180	TBCTL	Leitura / Escrita	Não utilizado	TBCLGRPx		CNTLx		Não utilizado	TBSSELx	
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura / Escrita	IDx		MCx		Não utilizado	TBCLR	TBIE	TBIFG
		Reset	0	0	0	0	0	0	0	0

TBCLGRPx - controle de agrupamento dos canais do *timer* B:

00 - cada canal carrega seu registrador TBCLx independentemente (símbolo **TBCLGRP_0**);

01 - agrupamento dos canais dois a dois (símbolo **TBCLGRP_1**):

- canal 0 independente;
- canal 1 + canal 2 (controle de atualização pelo canal 1);
- canal 3 + canal 4 (controle de atualização pelo canal 3);
- canal 5 + canal 6 (controle de atualização pelo canal 5);

10 - agrupamento dos canais três a três (símbolo **TBCLGRP_2**):

- canal 0 independente;
- canal 1 + canal 2 + canal 3 (controle de atualização pelo canal 1);
- canal 4 + canal 5 + canal 6 (controle de atualização pelo canal 4);

11 - agrupamento de todos os canais: a atualização é controlada pelo canal 1 (símbolo **TBCLGRP_3**).

CNTLx - largura de contagem do contador TBR:

00 - 16 *bits* (contagem máxima de 65.535) (símbolo **CNTL_0**);

01 - 12 *bits* (contagem máxima de 4.095) (símbolo **CNTL_1**);

10 - 10 *bits* (contagem máxima de 1.023) (símbolo **CNTL_2**);

11 - 8 *bits* (contagem máxima de 255) (símbolo **CNTL_3**).

TBSSELx - seleção da fonte de *clock* do *timer* B:

00 - TBCLK (*clock* externo) (símbolo **TBSSEL_0**);

01 - *clock* auxiliar (ACLK) (símbolo **TBSSEL_1**);

10 - *clock* secundário (SMCLK) (símbolo **TBSSEL_2**);

11 - TBCLK invertido (símbolo **TBSSEL_3**).

IDx - divisor do *clock* do timer B:

IDx	Fator de divisão	Símbolo
00	1	**ID_0**
01	2	**ID_1**
10	4	**ID_2**
11	8	**ID_3**

MCx - seleção do modo de funcionamento do *timer* A:

00 - parado (símbolo **MC_0**);

01 - contagem progressiva com módulo (0 até o valor de TBCL0) (símbolo **MC_1**);

10 - contínuo (0 até a contagem máxima (CNTLx) e depois reinicia do zero) (símbolo **MC_2**);

11 - crescente/decrescente (contagem de 0 até TBCL0 e depois até 0 novamente) (símbolo **MC_3**).

TBCLR - *reset* do *timer* B: setando esse *bit*, a contagem do TBR, do divisor de *clock* e a direção de contagem são apagadas. O *bit* retorna automaticamente a zero após o *reset* do *timer* (símbolo **TBCLR**).

TBIE - habilitação da interrupção de estouro de contagem do TBR:

0 - interrupção desabilitada;

1 - interrupção habilitada (símbolo **TBIE**).

TBIFG - sinalizador de interrupção do TBR:

0 - nenhuma interrupção pendente;

1 - o TBR estourou a contagem e há uma interrupção pendente (símbolo **TBIFG**).

5.6.8.2. TBR

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0190	TBR	Leitura / Escrita	TBRx							
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura / Escrita	TBRx							
		Reset	0	0	0	0	0	0	0	0

TBRx - 16 *bits* do registrador de contagem do timer B.

5.6.8.3. TBCCTLx

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0182	TBCCTL0	Leitura / Escrita	CMx		CCISx		SCS	CLLDx		CAP
0x0184	TBCCTL1									
0x0186	TBCCTL2	*Reset*	0	0	0	0	0	0	0	0
0x0188	TBCCTL3		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x018A	TBCCTL4	Leitura	OUTMODx			CCIE	CCI	OUT	COV	CCIFG
0x018C	TBCCTL5	Escrita					-			
0x018E	TBCCTL6	*Reset*	0	0	0	0	0	0	0	0

CMx - seleção do modo de captura:

00 - não realiza capturas (símbolo **CM_0**);

01 - captura na borda de subida (símbolo **CM_1**);

10 - captura na borda de descida (símbolo **CM_2**);

11 - captura em ambas as bordas (símbolo **CM_3**).

CCISx - seleção da entrada de captura/comparação:

00 - CCIxA (símbolo **CCIS_0**);

01 - CCIxB (símbolo **CCIS_1**);

10 - GND (símbolo **CCIS_2**);

11 - V_{CC} (símbolo **CCIS_3**).

SCS - sincronização da entrada de captura com o *clock* interno:

0 - captura assíncrona;

1 - captura síncrona (símbolo **SCS**).

CLLDx - seleção do evento de disparo da atualização do registrador TBCLx:

00 - o registrador TBCLx é atualizado após a escrita no TBCCRx (símbolo **CLLD_0**);

01 - o registrador TBCLx é atualizado quando o TBR conta até 0 (símbolo **CLLD_1**);

10 - o registrador TBCLx é atualizado quando o TBR conta até 0 (tanto no modo de contagem de módulo como na contagem contínua), ou quando o TBR conta até o valor de TBCL0 ou 0 (no modo crescente/decrescente) (símbolo **CLLD_2**);

11 - o registrador TBCLx é atualizado quando o TBR conta até o valor de TBCLx (símbolo **CLLD_3**).

CAP - seleção do modo de captura/comparação:

0 - modo de comparação;

1 - modo de captura (símbolo **CAP**).

OUTMODx - seleção do modo de comparação:

000 - a saída recebe o valor do *bit* OUT (símbolo **OUTMOD_0**);

001 - a saída é setada na comparação (símbolo **OUTMOD_1**);

010 - a saída é invertida na comparação e apagada ao atingir o valor do TBCL0 (símbolo **OUTMOD_2**);

011 - a saída é setada na comparação e apagada ao atingir o valor do TBCL 0 (símbolo **OUTMOD_3**);

100 - a saída é invertida a cada comparação (símbolo **OUTMOD_4**);

101 - a saída é apagada na comparação (símbolo **OUTMOD_5**);

110 - a saída é invertida na comparação e setada ao atingir o valor do TBCL 0 (símbolo **OUTMOD_6**);

111 - a saída é apagada na comparação e setada ao atingir o valor do TBCL 0 (símbolo **OUTMOD_7**).

CCIE - habilitação da interrupção de captura/comparação do canal:

0 - interrupção desabilitada;

1 - interrupção habilitada (símbolo **CCIE**).

CCI - entrada de captura/comparação: permite ler o estado do sinal da entrada de captura/comparação (símbolo **CCI**).

OUT - estado do sinal de saída (OUTx) do canal. No modo 0, esse *bit* controla diretamente o estado da saída de comparação:

0 - saída em nível lógico "0";

1 - saída em nível lógico "1" (símbolo **OUT**).

COV - indicador de *overflow* de captura, ou seja, uma segunda captura foi feita antes da leitura da primeira:

0 - não ocorreu *overflow* de captura;

1 - ocorreu um *overflow* de captura (símbolo **COV**).

CCIFG - sinalizador de interrupção de captura/comparação:

0 - nenhuma interrupção pendente;

1 - uma interrupção de captura/comparação está pendente (símbolo **CCIFG**).

5.6.8.4. TBCCRx

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0192	TBCCR0	Leitura	TBCCRx							
0x0194	TBCCR1	Escrita								
0x0196	TBCCR2	*Reset*	0	0	0	0	0	0	0	0
0x0198	TBCCR3		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x019A	TBCCR4	Leitura	TBCCRx							
0x019C	TBCCR5	Escrita								
0x019E	TBCCR6	*Reset*	0	0	0	0	0	0	0	0

TBCCRx - 16 *bits* do registrador de captura/comparação do canal x. Lembrando que nos modos 1 e 3 do *timer* o registrador TBCL0 funciona como registrador de módulo de contagem do TBR.

5.6.8.5. TBIV

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x011E	TB1IV	Leitura	0	0	0	0	0	0	0	0
		Escrita	-	-	-	-	-	-	-	-
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	0	0	0	0	TBIVx			0
		Escrita	-	-	-	-				-
		Reset	0	0	0	0	0	0	0	0

TBIV	Fonte de interrupção	*Flag* de interrupção	Prioridade
0x00	Nenhuma interrupção	-	
0x02	CCP canal 1	TBCCR1:CCIFG	Maior
0x04	CCP canal 2	TBCCR2:CCIFG	
0x06	CCP canal 3	TBCCR3:CCIFG	
0x08	CCP canal 4	TBCCR4:CCIFG	
0x0A	CCP canal 5	TBCCR5:CCIFG	
0x0C	CCP canal 6	TBCCR6:CCIFG	
0x0E	Estouro do *timer*	TBCTL:TBIFG	Menor

Tabela 5-15

5.7. Temporizador Básico (*Timer 1*)

O temporizador básico é formado por dois contadores de 8 *bits* destinados a aplicações distintas:

- O primeiro contador (BTCNT1) é utilizado para gerar o sinal de *clock* necessário para o módulo controlador de LCD.
- O segundo contador (BTCNT2) pode ser utilizado basicamente como um divisor de freqüências programável e com capacidade de gerar interrupções. A sua principal aplicação é a geração de interrupções periódicas da CPU, e pode funcionar como base de um sistema de relógio de tempo real (RTC).

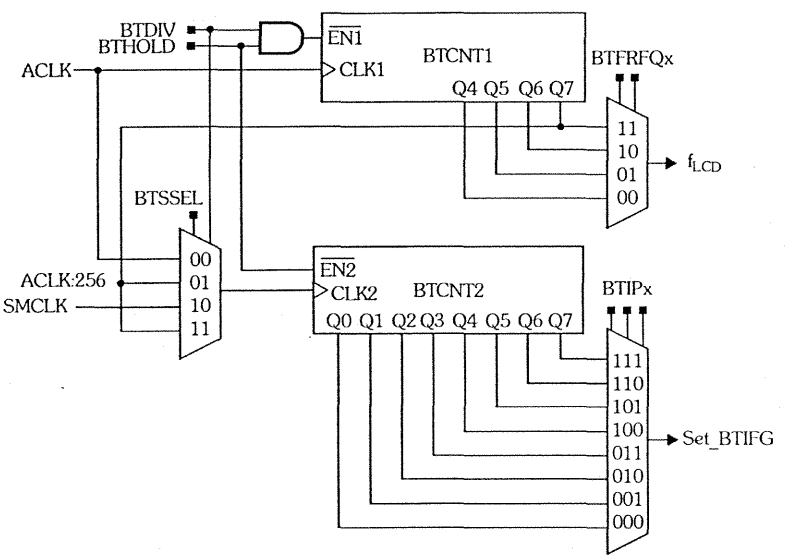


Figura 5-17

5.7.1. Contador 1

O contador BTCNT1 é utilizado para gerar o sinal de *clock* para o controlador de LCD e utiliza como fonte o sinal de *clock* auxiliar (ACLK).

Como podemos observar na figura 5-17, a freqüência de saída para o LCD (f_{LCD}) pode ser igual a ACLK/256, ACLK/128, ACLK/64 ou ACLK/32. Essa seleção é feita pelos *bits* BTFRFQx (registrador BTCTL).

A freqüência f_{LCD} pode ser calculada pela seguinte fórmula:

$$f_{LCD} = ACLK/x$$

sendo:

- ACLK é a freqüência do *clock* auxiliar;
- x é fator de divisão selecionado por BTFRFQx (32, 64, 128 ou 256).

Observe que o contador BTCNT1 é um registrador de 8 *bits* que pode ser plenamente acessado pelo *software* do usuário.

5.7.2. Contador 2

O contador BTCNT2 pode ser utilizado como um gerador de intervalos periódicos, especialmente em aplicações de medição de tempo como em um relógio de tempo real.

O *clock* do BTCNT2 pode ser originado de três possíveis fontes: *clock* auxiliar (ACLK), *clock* secundário (SMCLK) ou da saída do BTCNT1, em que obtemos uma freqüência igual a ACLK/256. Neste último caso, os dois contadores trabalham em cascata, atuando como um temporizador de 16 *bits*.

A seleção da fonte de *clock* do BTCNT2 é feita pelos *bits* BTSSEL e BTDIV, ambos localizados no registrador BTCTL.

A saída do contador gera o sinal de interrupção BTIFG, o qual corresponde ao *clock* do BTCNT2 dividido por um fator igual a 2, 4, 8, 16, 32, 64, 128 ou 256, selecionável por um demultiplexador 8 para 1. Os *bits* BTIPx selecionam qual dos oito fatores de divisão será utilizado.

O contador BTCNT2, assim como o BTCNT1, também é um registrador de 8 *bits* e pode ser plenamente acessado pelo *software* do usuário.

5.7.3. Interrupção do Temporizador Básico

O temporizador básico dispõe somente de uma única interrupção (BTIFG, registrador IFG2). O seu vetor de interrupção é o de número zero.

5.7.4. Registradores do Temporizador Básico

O temporizador básico dispõe de três registradores de 8 *bits*, mais dois registradores SFR em que estão localizados o controle e o *flag* de interrupção:

- **BTCTL** - registrador de controle do *timer* básico;
- **BTCNT1** - registrador de acesso à contagem do contador 1;
- **BTCNT2** - registrador de acesso à contagem do contador 2;
- **IFG2** - registrador em que se encontra localizado o *flag* BTIFG;
- **IE2** - registrador em que se encontra localizado o *bit* BTIE.

5.7.4.1. BTCTL

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0040	BTCTL	Leitura / Escrita	BTSSEL	BTHOLD	BTDIV	BTFRFQx		BTIPx		
		Reset	?	?	?	?	?	?	?	?

BTSSEL - juntamente com o *bit* BTDIV, seleciona a fonte de *clock* para o contador BTCNT2 (símbolo **BTSSEL**).

BTHOLD - parada de contagem do temporizador básico:

0 - ambos os contadores estão operacionais;

1 - a contagem do contador BTCNT2 é paralisada e a contagem do BTCNT1 é paralisada se BTDIV = 1 (símbolo **BTHOLD**).

BTDIV - divisor de *clock* do *timer* básico. Juntamente com o *bit* BTSSEL seleciona a fonte de *clock* do contador BTCNT2:

BTSSEL	BTDIV	Fonte de *clock* do BTCNT2	Símbolo
0	0	ACLK	BT_fCLK2_ACLK
0	1	ACLK/256 (via BTCNT1)	BT_fCLK2_ACLK_DIV256
1	0	SMCLK	BT_fCLK2_MCLK
1	1	ACLK/256 (via BTCNT1)	BT_fCLK2_ACLK_DIV256

BTFRFQx - seleção da freqüência de atualização do LCD:

00 - ACLK/32 (símbolo **BT_fLCD_DIV32**);

01 - ACLK/64 (símbolo **BT_fLCD_DIV64**);

10 - ACLK/128 (símbolo **BT_fLCD_DIV128**);

11 - ACLK/256 (símbolo **BT_fLCD_DIV256**).

BTIPx - freqüência de interrupção BTIFG:

000 - $f_{BTCNT2CLK}/2$ (símbolo **BT_fCLK2_DIV2**);

001 - $f_{BTCNT2CLK}/4$ (símbolo **BT_fCLK2_DIV4**);

010 - $f_{BTCNT2CLK}/8$ (símbolo **BT_fCLK2_DIV8**);

011 - $f_{BTCNT2CLK}/16$ (símbolo **BT_fCLK2_DIV16**);

100 - $f_{BTCNT2CLK}/32$ (símbolo **BT_fCLK2_DIV32**);

101 - $f_{BTCNT2CLK}/64$ (símbolo **BT_fCLK2_DIV64**);

110 - $f_{BTCNT2CLK}/128$ (símbolo **BT_fCLK2_DIV128**);

111 - $f_{BTCNT2CLK}/256$ (símbolo **BT_fCLK2_DIV256**).

5.7.4.2. BTCNT1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0046	BTCNT1	Leitura / Escrita	BTCNT1x							
		Reset	?	?	?	?	?	?	?	?

BTCNT1x - contagem do contador BTCNT1, responsável pela temporização do LCD.

5.7.4.3. BTCNT2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0047	BTCNT2	Leitura Escrita	BTCNT2x							
		Reset	?	?	?	?	?	?	?	?

BTCNT2x - contagem do contador BTCNT2.

5.7.4.4. IE2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0001	IE2	Leitura Escrita	**BTIE**	-	UTXIE1	URXIE1	-	-	-	-
		Reset	0	0	0	0	0	0	0	0

BTIE - habilitação da interrupção do *timer* básico 1 (símbolo **BTIE**).

5.7.4.5. IFG2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0003	IFG2	Leitura Escrita	**BTIFG**		UTXIFG1	URXIFG1				
		Reset	0	0	1	0	0	0	0	0

BTIFG - sinalizador da interrupção *timer* básico 1 (símbolo **BTIFG**).

5.7.5. Exemplo de Utilização

No exemplo seguinte, configuramos o *timer* básico para atuar como um temporizador de intervalo. Operando a partir do ACLK, com uma freqüência de 32.768Hz, o contador 2 do *timer* básico gera uma interrupção por segundo.

No corpo da função de tratamento da interrupção do *timer*, escrevemos um programa que pode ser utilizado para implementar um relógio, contando horas, minutos e segundos.

O programa principal permanece em modo LPM3, o que reduz enormemente o consumo de energia, permitindo a implementação de sistemas à bateria com grande autonomia.

```
#include "io430x44x.h"
#include "intrinsics.h"

unsigned char horas, minutos, segundos;

// estas definições são necessárias, pois não fazem parte do arquivo
// io430x44x.h
#define BTIP0 1
#define BTIP1 2
#define BTIP2 4

//*********************************************************************
// Função: trata_BT
//*********************************************************************
// Esta função trata a interrupção do contador 2 do timer básico. Ele
// está configurado para dividir o ACLK por 256 * 128 = 32768
// resultando uma interrupção por segundo (quando utilizando um
// cristal de 32.768Hz no LFXT1).
//*********************************************************************
#pragma vector = BASICTIMER_VECTOR
__interrupt void trata_BT(void)
{
   segundos++;            // incrementa segundos
   if (segundos>59)       // se segundos maior que 59
   {
      segundos = 0;       // zera segundos
      minutos++;  // incrementa minutos
      if (minutos>59)     // se minutos maior que 59
      {
         minutos = 0;     // zera minutos
         horas++; // incrementa horas
         if (horas>23) horas = 0;       // se horas maior que 23
                                        // zera as horas
      }
   }
}
void main(void)
{
   unsigned int temp;
   WDTCTL = WDTPW+WDTHOLD;        // desliga o watchdog
   // configura o timer básico, o contador 2 opera dividindo
   // o ACLK por 256 * 128 = 32768
   BTCTL = BTDIV + BT_fCLK2_DIV128;     // ACLK / 32768
   IE2 = BTIE;    // habilita interrupção do timer básico
   segundos = minutos = horas = 0;
   // configura todos os pinos como saída, para reduzir o consumo
   P1DIR = P2DIR = P3DIR = P4DIR = P5DIR = P6DIR = 0xFF;
   __enable_interrupt(); // habilita as interrupções
   __low_power_mode_3(); // entrada no modo LMP3
   while (1);            // aguarda a interrupção
}
```

Exemplo 5-8

5.8. USART - Modo Assíncrono

Alguns modelos MSP430 possuem uma interface serial síncrona/assíncrona universal (USART) (os modelos 12x, 13x, 15x, 42x e 43x possuem uma USART (USART0), enquanto os modelos 14x, 16x e 44x possuem duas USARTs independentes (USART0 e USART1)).

O modo assíncrono, disponível em todos os modelos de USART, permite a comunicação entre o microcontrolador e outros equipamentos ou dispositivos, tais como: computadores, outros microcontroladores, dispositivos de comunicação, etc.

A USART implementada nos MSPs suporta tamanhos de caractere de 7 ou 8 *bits*, geração e detecção automática de paridade e seleção do número de *bits* de parada (1 ou 2).

Também estão implementados circuitos para detecção de erros de comunicação e para o suporte a endereçamento em sistemas multiprocessados.

O suporte a endereçamento pode ser implementado utilizando dois protocolos diferentes: detecção de linha inativa (*Idle-Line Detection*) ou utilizando *bits* de endereçamento. Esses protocolos serão vistos mais adiante.

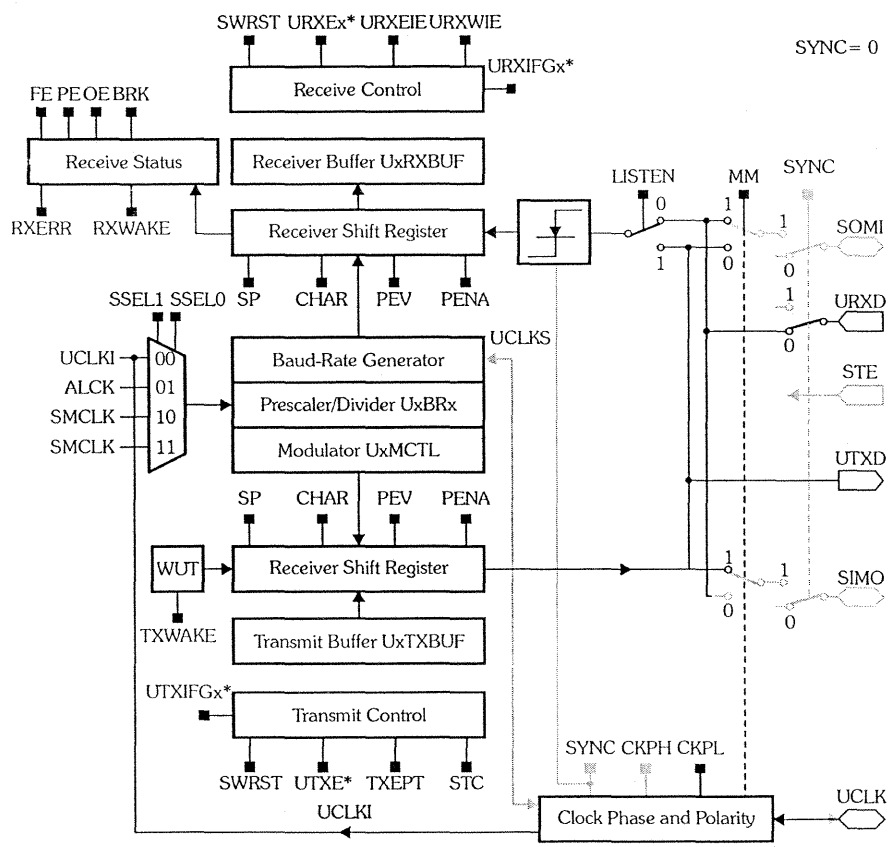


Figura 5-18

A USART possui também dois registradores de deslocamento, separados e independentes, para transmissão e recepção. Além disso, um circuito gerador de *clock* (*baud-rate*) permite a

obtenção de diversas velocidades de comunicação, a partir de uma das fontes de *clock* disponíveis (SMCLK, ACLK ou UCLKI (entrada de *clock* da USART)).

O formato de transmissão pode ser visualizado na figura 5-19:

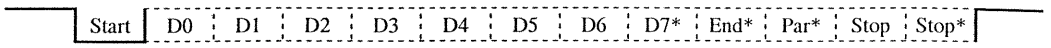


Figura 5-19

O formato de transmissão de um caractere é iniciado por um *bit* de partida (*start*), seguido do *bit* LSB do caractere e em seqüência os demais *bits*, até o MSB. A USART suporta caracteres de 7 ou 8 *bits*. Essa opção é feita pelo *bit* CHAR no registrador UxCTL.

Após o *bit* MSB do caractere, seguem-se um *bit* de endereçamento *(End)* (opcional), um *bit* de paridade (Par) (opcional), um *bit* de parada (*Stop*) e eventualmente um segundo *bit* de parada (*Stop*) (opcional).

5.8.1. Gerador de *Baud-Rate*

Antes de continuarmos com a transmissão e recepção da USART, é importante conhecermos o mecanismo responsável pela obtenção do sinal de *clock* para os registradores de deslocamento de transmissão e recepção.

O gerador de *baud-rate* (BRG) é composto por um divisor de 16 *bits*, associado a um modulador (como o utilizado no DCO). Esse conjunto permite realizar divisões fracionárias do sinal de *clock* de entrada e possibilita maior precisão na freqüência da taxa de sinalização (*baud-rate*).

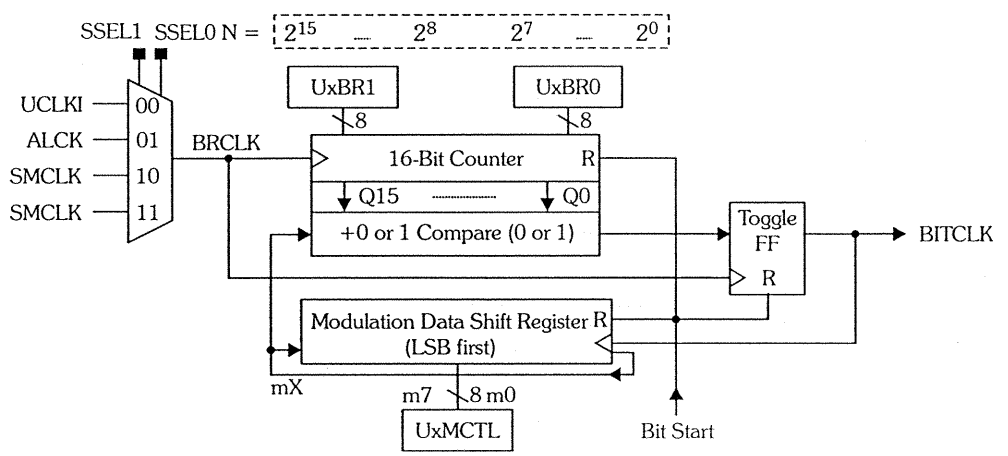


Figura 5-20

O funcionamento do BRG é o seguinte: o sinal de *clock* proveniente da fonte selecionada (*bits* SSELx, registrador UxTCTL) é aplicado a um contador de 16 *bits*, que conta de zero até o valor carregado em UxBR (UxBR1 e UxBR0 concatenados, em que UxBR1 corresponde aos 8 *bits* mais significativos e UxBR0 aos 8 *bits* menos significativos). Este é o fator de divisão inteiro do sinal de *clock* do BRG.

Adicionalmente, um *bit* do modulador é adicionado ao valor de UxBR, de forma a aproximar o resultado da divisão do valor fracionário desejado. Um *bit* do modulador é adicionado para cada *bit* transmitido. O tempo do *bit* de *start* é obtido a partir da soma do valor UxBR com o *bit* LSB do UxMCTL, o *bit* D0 é obtido da soma de UxBR com o *bit* 1 do UxMCTL e assim por diante. Após atingido o último *bit* do UxMCTL, utiliza-se novamente o primeiro e assim por diante.

Na prática, esse processo nada mais é do que uma divisão fracionária, em que a parte inteira é obtida dos registradores UxBR1 e UxBR0 e a parte fracionária é obtida do UxMCTL. O sinal de *clock* assim obtido é chamado de BITCLK e é utilizado pelos registradores de deslocamento de transmissão e recepção da USART.

A velocidade de comunicação pode ser calculada pelo uso da fórmula seguinte:

$$Baud-rate = \frac{BRCLK}{N} = \frac{BRCLK}{UxBR + \frac{1}{N}\sum_{i=0}^{n-1} m_i}$$

sendo:

- N - fator de divisão desejado
- UxBR - valor concatenado dos registradores UxBR1 e UxBR0
- i - posição do *bit* de modulação
- n - número total de *bits* no caractere
- m_i - estado do *bit* de modulação correspondente

O cálculo do valor ideal de UxMCTL é um processo relativamente complexo, mas que pode ser facilitado enormemente pelo uso da planilha eletrônica inclusa nos arquivos disponíveis para *download*.

Um detalhe importante é que o desvio de freqüência varia conforme a posição do *bit* transmitido/recebido. Neste caso, utilizamos o maior erro como sendo o de desvio de velocidade de comunicação.

O percentual de erro de um determinado *bit* pode ser calculado pela seguinte fórmula:

$$Erro(\%) = (\frac{baud-rate}{BRCLK} * ((j+1) * UxBR + \sum_{i=0}^{n-1} m_i) - (j+1)) * 100$$

sendo:

- *baud-rate* - taxa de sinalização (velocidade de comunicação)
- BRCLK - *clock* de entrada do BRG
- j - posição do *bit* no caractere
- UxBR - valor concatenado dos registradores UxBR1 e UxBR0
- i - posição do *bit* de modulação
- n - número total de *bits* no caractere
- m_i - estado do *bit* de modulação correspondente

5.8.1.1. Tabelas de Configuração para Velocidades Típicas

As tabelas seguintes apresentam a configuração dos registradores do gerador de *baud-rate* para a obtenção de diversas velocidades padronizadas em algumas freqüências de operação do *chip*.

Velocidade	Clock							
	32768Hz				500KHz			
	UxBR1	UxBR0	UxMCTL	Erro Máx. (%)	UxBR1	UxBR0	UxMCTL	Erro Máx. (%)
1200	0x00	0x27	0x12	-2,78	0x01	0xA0	0x6D	-0,16
2400	0x00	0x0D	0x6D	-3,91	0x00	0xD0	0x92	0,32
4800	0x00	0x06	0xFB	10,74	0x00	0x68	0x04	-0,64
9600	0x00	0x03	0x4A	-21,09	0x00	0x34	0x20	0,96
19200	-	-	-	-	0x00	0x1A	0x00	-1,6
38400	-	-	-	-	0x00	0x0D	0x00	-1,6
57600	-	-	-	-	0x00	0x08	0x6D	-9,28
76800	-	-	-	-	0x00	0x06	0x55	7,52
115200	-	-	-	-	0x00	0x04	0x92	13,76
230400	-	-	-	-	0x00	0x02	0x04	-32,32

Tabela 5-16

Velocidade	Clock							
	1MHz				2MHz			
	UxBR1	UxBR0	UxMCTL	Erro Máx. (%)	UxBR1	UxBR0	UxMCTL	Erro Máx. (%)
1200	0x03	0x41	0x92	0,08	0x06	0x82	0x6D	-0,04
2400	0x01	0xA0	0x6D	-0,16	0x03	0x41	0x92	0,08
4800	0x00	0xD0	0x92	0,32	0x01	0xA0	0x6D	-0,16
9600	0x00	0x68	0x04	-0,64	0x00	0xD0	0x92	0,32
19200	0x00	0x34	0x20	0,96	0x00	0x68	0x04	-0,64
38400	0x00	0x1A	0x00	-1,60	0x00	0x34	0x20	0,96
57600	0x00	0x11	0x52	2,72	0x00	0x22	0xDD	1,44
76800	0x00	0x0D	0x00	-1,60	0x00	0x1A	0x00	-1,60
115200	0x00	0x08	0x6D	-9,28	0x00	0x11	0x52	2,72
230400	0x00	0x04	0x92	13,76	0x00	0x08	0x6D	-9,28

Tabela 5-17

Velocidade	Clock							
	4MHz				8MHz			
	UxBR1	UxBR0	UxMCTL	Erro Máx. (%)	UxBR1	UxBR0	UxMCTL	Erro Máx. (%)
1200	0x0D	0x05	0x92	0,02	0x1A	0x0A	0x6D	-0,01
2400	0x06	0x82	0x6D	-0,04	0x0D	0x05	0x92	0,02
4800	0x03	0x41	0x92	0,08	0x06	0x82	0x6D	-0,04
9600	0x01	0xA0	0x6D	-0,16	0x03	0x41	0x92	0,08
19200	0x00	0xD0	0x92	0,32	0x01	0xA0	0x6D	-0,16
38400	0x00	0x68	0x04	-0,64	0x00	0xD0	0x92	0,32
57600	0x00	0x45	0xAA	0,80	0x00	0x8A	0xEF	-0,32
76800	0x00	0x34	0x20	0,96	0x00	0x68	0x04	-0,64
115200	0x00	0x22	0xDD	1,44	0x00	0x45	0xAA	0,80
230400	0x00	0x11	0x52	2,72	0x00	0x22	0xDD	1,44

Tabela 5-18

No *site* da editora, o leitor poderá baixar uma planilha eletrônica que permite calcular automaticamente o valor de carga dos registradores para qualquer velocidade de *clock* e de comunicação.

5.8.2. Configuração da USART

Antes de iniciar o processo de comunicação utilizando a USART, é necessário configurá-la.

A seqüência de configuração correta e recomendada pelo fabricante é:

1. Configurar os pinos de Tx e Rx da USART para a função alternativa (veja a tabela 5-19);
2. Ativar o *reset* da USART (*bit* SWRST do registrador UxCTL);
3. Configurar os registradores da USART:
   - UxCTL: SYNC = 0, SWRST = 0 e os demais *bits* conforme o modo desejado;
   - UxTCTL: fonte de *clock* e a sua polaridade (*bits* SSELx e CKPL);
   - UxRCTL: controle do modo de recepção;
   - Velocidade de comunicação (registradores UxBR0, UxBR1 e UxMCTL);
4. Habilitar a USART (*bits* UTXEx e URXEx, nos registradores ME1 ou ME2, conforme a USART desejada);
5. Desativar o *reset* da USART (*bit* SWRST = 0);
6. Caso se deseje utilizar as interrupções da USART, habilitá-las pelos *bits* UTXIEx e/ou URXIEx (registradores IE1 ou IE2, conforme a USART desejada). **Lembre-se: as interrupções somente podem ser ativadas com SWRST=0!**

O formato de transmissão pode ser configurado no registrador UxCTL. Podemos selecionar entre a transmissão de 7 (CHAR = 0) ou 8 (CHAR = 1) *bits* de dados, com paridade (PENA=1) ou sem (PENA=0) e adicionalmente, podemos incluir um segundo *bit* de parada (SPB = 1).

Lembre-se de que a ativação do sinal de *reset* da USART (UxCTL:SWRST =1) faz com que os *bits* URXIEx, UTXIEx, URXIFGx, RXWAKE, TXWAKE, RXERR, BRK, PE, OE e FE sejam apagados e os *bits* UTXIFGx e TXEPT sejam setados.

5.8.3. Transmissão Serial

A transmissão serial de um caractere é um processo extremamente simples. Após a configuração inicial da USART e estando habilitada (UTXEx:MEx = 1), a simples escrita de um dado no *buffer* de transmissão (registrador UxTXBUF) provoca o início da transmissão.

Iniciada a transmissão, recomenda-se não alterar o conteúdo do *buffer* de transmissão até que a USART esteja pronta para uma nova transmissão. Essa situação é sinalizada pelo *flag* de interrupção IFGx:UTXIFGx que, se estiver setado, um novo dado pode ser escrito no UxTXBUF.

Quando a transmissão corrente é totalmente concluída, ou seja, todos os *bits* do caractere, além dos *bits* adicionais e de parada, foram devidamente transmitidos pelo registrador de deslocamento de transmissão, o *bit* TXEPT (registrador UxTCTL) é setado.

É recomendável somente desabilitar a USART após concluída a transmissão corrente (UxTCTL:TXEPT = 1).

Caso seja escrito um dado no *buffer* de transmissão com a USART desabilitada (UTXEx:MEx = 0), nenhuma transmissão ocorre. No entanto, tão logo ela seja habilitada, o dado presente no *buffer* será carregado no registrador de deslocamento de transmissão e o processo de transmissão terá início.

Se o usuário desejar, pode habilitar a interrupção da USART (UTXIEx:IEx = 1), permitindo que, cada vez que o *buffer* de transmissão estiver vazio (o caractere tenha sido inteiramente carregado no registrador de deslocamento de transmissão), o *flag* IFGx:UTXIFGx possa interromper a CPU, permitindo que a rotina de tratamento de interrupção execute as operações desejadas, como, por exemplo, carregar um novo caractere no *buffer* de transmissão.

5.8.4. Recepção Serial

A recepção de um dado serialmente é um processo assíncrono em relação à CPU.

Inicialmente, a USART permanece monitorando a linha e quando ocorre uma transição marca-espaço (de nível "1" para nível "0"), o circuito de recepção mede o período do sinal, de forma a detectar o *bit* de *start* (partida).

O *bit* de partida possui a duração igual ao tempo de um *bit* da USART. Caso o circuito de recepção não o valide, a USART retorna à sua condição inicial, esperando pelo *bit* de partida.

A USART utiliza um sistema de voto majoritário, em que o sinal de entrada é avaliado no centro do período do sinal de *clock* BITCLK. Nesse instante, três amostras consecutivas do sinal de recepção são lidas e comparadas. O estado lógico do sinal de entrada é definido como o estado lido pela maioria dessas amostragens, ou seja, em duas delas.

Detectado um *bit* de partida válido, os *bits* seguintes são lidos e deslocados para o registrador de deslocamento de recepção.

Após a recepção dos 7 ou 8 *bits* de dados, além dos *bits* adicionais que eventualmente existirem, bem como o(s) *bit*(s) de parada, o circuito de detecção de erros da USART verifica a existência de um dos possíveis erros de recepção:

- UxRCTL:FE - quadro (*framing error*) - quando um *bit* de parada em nível "0" é detectado.
- UxRCTL:PE - paridade (*parity error*) - quando a paridade do caractere lido é diferente da calculada.
- UxRCTL:OE - sobrescrita (*receive overrun error*) - quando um novo caractere é recebido antes da leitura do anterior (armazenado no UxRXBUF).
- UxRCTL:BRK - *break condition* - quando a linha permanece em "0" por 10 ou mais tempos de *bit*.

Caso a recepção tenha ocorrido sem erros, o *flag* IFGx:URXIFGx é setado, indicando que um novo caractere foi recebido e está disponível no *buffer* de recepção (UxRXBUF).

Quando qualquer um desses erros é detectado, o seu respectivo *flag* é setado, assim como o *bit* RXERR (registrador UxRCTL). Neste caso, a ativação do *flag* IFGx:URXIFGx vai depender do *bit* URXEIE (registrador UxRCTL) que controla como ocorre a recepção de caracteres:

Com UxRCTL:URXEIE = 0, a recepção de um caractere contendo erros é descartada (ele não é escrito no *buffer* de recepção) e o *flag* IFGx:URXIFGx não é setado.

Com UxRCTL:URXEIE = 1, a recepção de um caractere contendo erros ocorre normalmente. O caractere contendo erros é armazenado no UxRXBUF e o *flag* IFGx:URXIFGx é setado. A leitura do *flag* UxRCTL:RXERR permite verificar a existência de erros de recepção.

Note que, quando algum dos *flags* de erro de recepção é ativado, ele permanecerá ativado até que o usuário apague-os.

Os *flags* de erro de recepção também são apagados pela simples leitura do *buffer* de recepção (UxRXBUF). Sendo assim, é importante ler os *flags* de erro **antes** da leitura do caractere recebido.

Uma funcionalidade adicional do circuito de recepção da USART é a capacidade de ativação da CPU na detecção da borda de descida do *bit* de partida (*start bit*). Essa operação é controlada pelo *bit* URXSE (registrador UxTCTL).

Quando ativada (UxTCTL:URXSE = 1), ao detectar a primeira borda de descida do sinal da linha de recepção (o início do *bit* de partida), caso a interrupção de recepção da USART esteja habilitada (UxRCTL:URXIEx = 1) e o GIE = 1, uma interrupção de recepção de caractere é disparada, **porém o flag IFGx:URXIFGx permanece apagado**.

Note que a USART inclui um circuito supressor de ruídos que impedirá que qualquer sinal com período inferior a aproximadamente 300ns provoque o disparo da interrupção.

Disparada a interrupção, a rotina de tratamento de interrupção deve providenciar a leitura do estado do *flag* IFGx:URXIFGx, o que significa que um caractere foi recebido e encontra-se disponível no UxRXBUF.

No caso de o *flag* estar apagado, trata-se da detecção da borda de descida do *bit* de partida. Neste caso, a RTI deve providenciar a eventual saída de um modo de baixa potência e comutar o *clock* da USART para uma fonte cuja freqüência seja estável e conhecida. É recomendável utilizar o DCO, devido à sua característica de partida extremamente rápida (menor que 6μs).

Além disso, o *bit* UxTCTL:URXSE deve ter o seu estado invertido, o que faz com que o evento causador da interrupção (o sinal interno conhecido como URXS) seja apagado. A inversão do URXSE também permite que a facilidade de detecção da borda de descida do *bit* de partida continue habilitada.

5.8.5. Endereçamento *Idle-Line*

Quando se pretende realizar a comunicação entre três ou mais dispositivos, utilizando uma mesma via de comunicação serial, é necessário utilizar um protocolo que permita identificar o dispositivo com o qual deseja se comunicar.

As USARTs dos MSP430 incluem o suporte a dois protocolos distintos: o endereçamento por inatividade da linha (*idle-line*) e o endereçamento por *bit* adicional de identificação.

A seleção de uma modalidade ou outra é feita pelo *bit* MM (registrador UxCTL): com MM = 0, temos o modo de endereçamento por inatividade da linha e com MM = 1, temos o modo de endereçamento por *bit*.

No controle de endereçamento por inatividade da linha, a especificação do dispositivo alvo (o seu endereço) é feita por um controle de temporização: o primeiro dado recebido após um tempo de inatividade de linha correspondente a 10 tempos de *bit* é considerado um campo de endereço. Os demais dados subseqüentes são considerados caracteres de dados úteis.

Na figura 5-21, podemos observar o funcionamento desse modo de endereçamento: os campos "E" correspondem aos campos de endereço e os campos "D" são os campos de dados. Cada bloco de dados possui como primeiro caractere um endereço "E" e o tempo entre cada caractere não pode ser superior a dez tempos de *bit* (tempo "a" na figura). Um tempo de inatividade de linha superior a dez ciclos de *bit* (tempo "b" na figura) faz com que o receptor considere o primeiro caractere subseqüente como um campo de endereço.

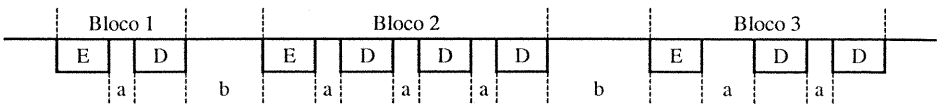


Figura 5-21

A transmissão de dados, neste modo de endereçamento, deve respeitar atraso máximo de dez tempos de *bit* entre um caractere e outro (este tempo começa a contar a partir do primeiro *bit* de parada do caractere).

A USART inclui um mecanismo que facilita a obtenção da temporização correta para o caractere de endereçamento. Basta observar os seguintes passos:

1. O *bit* UxTCTL:TXWAKE deve ser setado.
2. Escreve-se um valor qualquer no *buffer* de transmissão (UxTXBUF). Isso provoca o início de uma transmissão fictícia em que nenhum dado será efetivamente transmitido, porém o tempo correspondente a exatamente 11 tempos de *bit* será gasto na operação. Após essa operação, o *bit* TXWAKE é automaticamente apagado pelo *hardware* da UART.
3. Escreve-se o endereço desejado no *buffer* de transmissão. Esse dado será transmitido e como está precedido de um período de inatividade de linha superior a dez tempos de *bit*, será considerado um campo de endereço.

Observe que qualquer atraso superior a dez tempos de *bit*, entre a transmissão de um dado e o seguinte, fará com que o receptor interprete que o dado recebido após o atraso é um endereço. O *software* do transmissor deve obedecer estritamente à temporização mínima entre um caractere e outro.

A recepção de dados utilizando o endereçamento por inatividade da linha é feita simplesmente setando o *bit* UxRCTL:URXWIE que possui a finalidade de controlar a forma como ocorre a recepção de dados da UART: se ele estiver apagado, todos os dados recebidos são checados contra a existência de erros e (eventualmente) transferidos para o *buffer* de recepção (UxRXBUF).

Quando o *bit* UxRCTL:URXWIE está setado, somente quando um caractere é validado como sendo de endereço, ele é armazenado no *buffer* de recepção (gerando assim uma interrupção de recepção de dado, caso ela esteja ativada). Desta forma o *software* do usuário pode verificar se o endereço recebido corresponde ao seu endereço. Caso positivo, o *bit* URXWIE deve ser apagado para continuar a receber os dados.

Com o *bit* URXWIE apagado todos os caracteres recebidos pela USART serão considerados dados, no entanto o *software* do usuário deve monitorar o *bit* UxRCTL:RXWAKE a cada caractere recebido, pois ele reflete o tipo de caractere armazenado no *buffer* de recepção: caso RXWAKE = "1", o caractere recebido é um endereço e caso RXWAKE = "0", o caractere recebido é um dado.

Assim, ao receber um caractere de endereço, ele deve ser novamente validado e caso seja diferente do endereço programado no *software*, o *bit* URXWIE deve ser setado, de forma a ignorar os caracteres subseqüentes e aguardar um novo endereço.

5.8.6. Endereçamento por *bit*

Nessa modalidade de operação, um *bit* adicional é inserido após o último *bit* do caractere. Esse *bit* adicional ("End" na figura 5-19) possui a finalidade de especificar se o caractere é um endereço ("End" = 1) ou um dado válido ("End" = 0).

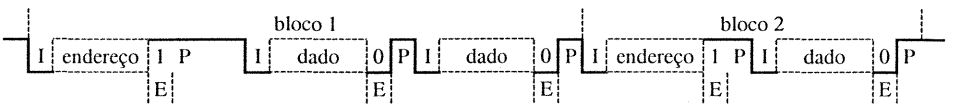


Figura 5-22

Na figura 5-22, podemos observar o funcionamento dessa modalidade de endereçamento. Como cada caractere inclui um *bit* ("E") para indicar a natureza do seu conteúdo (endereço se E = 1 e dado se E = 0), a temporização entre um caractere e o seguinte é irrelevante.

A transmissão nessa modalidade é feita da seguinte forma: para transmitir um endereço, basta setar o *bit* UxTCTL:TXWAKE e em seguida escrever o valor correspondente ao endereço no registrador UxTXBUF. Após o início da transmissão, o *hardware* da UART apaga automaticamente o *bit* TXWAKE.

Para transmissão de um dado, basta escrevê-lo no *buffer* de transmissão (UxTXBUF), mantendo o *bit* TXWAKE em nível "0".

Observe que a linha pode permanecer inativa após a transmissão do endereço ou de um dado, já que a temporização entre os caracteres, neste modo, não é relevante.

A recepção de dados utilizando o endereçamento por *bit* é muito similar à do modo anterior.

O *bit* UxRCTL:URXWIE é utilizado para especificar o tipo de caractere que pode gerar a interrupção de recepção de dados (IFGx:URXIFGx) e será armazenado no *buffer* de recepção. Com URXWIE = 1 somente os caracteres recebidos com o *bit* de endereço setado são verificados contra erros e armazenados no *buffer* de recepção (UxRXBUF). Os caracteres de dados (*bit* de endereço igual a zero) são descartados.

Com URXWIE = 0 todos os caracteres recebidos, independentemente do estado do seu *bit* de endereço, são testados contra erros e armazenados no *buffer* de recepção, setando também o *flag* de recepção de dados da USART (URXIFGx).

Após a recepção de um endereço, o *software* do usuário deve validá-lo e caso coincida com o seu endereço armazenado em algum ponto do programa, o *bit* URXWIE deve ser apagado de forma que a UART possa continuar a receber o restante dos dados.

Com o *bit* URXWIE apagado todos os caracteres recebidos pela USART serão considerados dados, no entanto o *software* do usuário deve monitorar o *bit* UxRCTL:RXWAKE a cada caractere recebido, pois ele reflete o tipo de caractere armazenado no *buffer* de recepção: caso RXWAKE = 1, o caractere recebido é um endereço e caso RXWAKE = 0, o caractere recebido é um dado.

Assim, ao receber um caractere de endereço, ele deve ser novamente validado e caso seja diferente do endereço previamente configurado no programa, o *bit* URXWIE deve ser setado, de forma a ignorar os caracteres subseqüentes e aguardar um novo caractere de endereço.

5.8.7. Interrupções da USART

A USART possui duas fontes de interrupção distintas: a transmissão de caractere (*flag* IFGx:UTXIFGx) e a recepção de caractere (*flag* IFGx:URXIFGx).

O *flag* UTXIFGx é setado sempre que o dado presente no *buffer* de transmissão tenha sido completamente transferido para o registrador de deslocamento do transmissor. O apagamento desse *flag* pode ser feito de uma das seguintes formas:

1. Pela chamada automática da rotina de tratamento dessa interrupção;
2. Pela escrita de um novo caractere no registrador UTXBUF;
3. Pela escrita do valor "0" no *flag*.

Observe que após um *reset* do sistema (PUC) ou da USART (UxCTL:SWRST = 1), o *flag* IFGx:UTXIFGx é automaticamente setado (indicando que a USART está pronta para receber um novo caractere) e a interrupção desabilitada (*bit* IEx:UTXIEx = 0).

O *flag* URXIFGx é setado por um dos seguintes eventos:

1. Recepção de um caractere de endereço (quando o *bit* UxRCTL:URXWIE = 1).
2. Recepção de um caractere de endereço ou de dado (quando o *bit* URXWIE = 0).

Além dos dois eventos citados, a interrupção de recepção de caractere pode ser disparada também na detecção da borda de descida do *bit* de partida (quando UxTCTL:URXSE = 1). Neste caso, porém, o *flag* URXIFGx não é setado.

O apagamento do *flag* pode ocorrer de uma das seguintes formas:

1. Pelo desvio automático para a rotina de tratamento da interrupção de recepção de caractere (o apagamento automático do *flag*, neste caso, ocorre somente se o *bit* URXSE = 0);
2. Pela leitura do *buffer* de recepção (UxRXBUF);
3. Pela escrita do valor "0" no *flag*.

5.8.8. Conexões da USART

Os sinais da(s) USART(s) estão disponíveis nos seguintes pinos:

Sinal	Pino
UTXD0	P3.4 (família 1xx) P2.4 (família 4xx)
URXD0	P3.5 (família 1xx) P2.5 (família 4xx)
UTXD1	P3.6 (1xx) P4.0 (44x)
URXD1	P3.7 (1xx) P4.1 (44x)

Tabela 5-19

5.8.9. Registradores da USART

A USART, quando opera no modo assíncrono, utiliza os seguintes registradores para o seu funcionamento:

- **UxCTL** - para configuração geral da USART;
- **UxTCTL** - para configuração do transmissor da USART;
- **UxRCTL** - para configuração do receptor da USART;
- **UxMCTL, UxBR0** e **UxBR1** - para configuração do gerador de *baud-rate*;
- **UxRXBUF** - para armazenamento do último caractere recebido;
- **UxTXBUF** - para armazenamento do caractere a ser transmitido;
- **MEx** - para habilitação da USART;
- **IEx** - para habilitação das interrupções da USART;
- **IFGx** - para sinalização das interrupções da USART;
- **PxSEL** - seleção da função alternativa para os pinos da USART.

5.8.9.1. UxCTL

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0070 0x0078	U0CTL U1CTL	Leitura Escrita	PENA	PEV	SPB	CHAR	LISTEN	SYNC	MM	SWRST
		Reset	0	0	0	0	0	0	0	1

PENA - habilitação da gerador/verificador de paridade:
0 - paridade desabilitada;
1 - paridade habilitada, a paridade é gerada na transmissão e também verificada na recepção (símbolo **PENA**).

PEV - seleção da paridade (utilizado somente se a paridade estiver ligada (PENA = 1):
0 - paridade par;
1 - paridade ímpar (símbolo **PEV**).

SPB - seleção do *bit* de parada para a transmissão (a recepção é feita sempre com um *bit* de parada):
0 - um *bit* de parada;
1 - dois *bits* de parada (símbolo **SPB**).

CHAR - largura do caractere (7 ou 8 *bits*):
0 - 7 *bits* (o MSB do registrador UxTXBUF é desprezado);
1 - 8 *bits* (símbolo **CHAR**).

LISTEN - habilitação do *loopback* (a transmissão e a recepção são internamente conectadas):
0 - modo normal;
1 - modo *loopback*, a saída da USART (TX) é internamente conectada à recepção. Todos os dados transmitidos são recebidos pela etapa de recepção (símbolo **LISTEN**).

SYNC - seleção do modo síncrono:
0 - modo assíncrono;
1 - modo síncrono (símbolo **SYNC**).

MM - seleção do modo multiprocessador:

0 - modo de endereçamento por inatividade da linha;

1 - modo de endereçamento utilizando *bit* de endereço (símbolo **MM**).

SWRST - ativação do *reset* da USART:

0 - USART no modo normal;

1 - USART no modo *reset*. Os bits URXIEx, UTXIEx, URXIFGx, RXWAKE, TXWAKE, RXERR, BRK, PE, OE e FE são apagados e os bits UTXIFGx e TXEPT são setados (símbolo **SWRST**).

5.8.9.2. UxTCTL

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0071	U0TCTL	Leitura	Não utilizado	CKPL	SSELx		URXSE	TXWAKE	Não utilizado	TXEPT
0x0079	U1TCTL	Escrita								
		Reset	0	0	0	0	0	0	0	1

CKPL - seleção da polaridade de *clock* para modo síncrono:

0 - o *clock* interno (UCLKI) é igual ao *clock* externo (UCLK);

1 - o *clock* interno (UCLKI) é igual ao *clock* externo (UCLK) invertido (símbolo **CKPL**).

SSELx - seleção da fonte de *clock* para o gerador de *baud-rate* (símbolo **SSEL1** e **SSEL0**):

00 - UCLKI ;

01 - ACLK;

10 - SMCLK;

11 - SMCLK.

URXSE - habilitação da opção de ativação da USART na detecção de uma transição da linha de recepção:

0 - desligado;

1 - ligado (símbolo **URXSE**).

TXWAKE - seleção do tipo de caractere a ser transmitido:

0 - o próximo caractere a ser transmitido é um dado;

1 - o próximo caractere a ser transmitido é um endereço (símbolo **TXWAKE**).

TXEPT - sinalizador de término de transmissão:

0 - há informação sendo transmitida ou aguardando no *buffer* UxTXBUF;

1 - não há informação sendo transmitida e o *buffer* UxTXBUF está vazio (ou a USART está em modo de *reset*, com SWRST = 1) (símbolo **TXEPT**).

5.8.9.3. UxRCTL

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0072	U0RCTL	Leitura	FE	PE	OE	BRK	URXEIE	URXWIE	RXWAKE	RXERR
0x007A	U1RCTL	Escrita								
		Reset	0	0	0	0	0	0	0	0

FE - sinalizador de erro de quadro (detecta quando o *bit* de parada está baixo):

0 - sem erros;

1 - um caractere foi recebido com erro de quadro (símbolo **FE**).

PE - sinalizador de erro de paridade (somente ativo quando PENA = 1, caso a paridade esteja desativada, esse *bit* está sempre em zero):

0 - sem erros;

1 - a paridade calculada é diferente da recebida (símbolo **PE**).

OE - sinalizador de erro de sobrescrita (indica que um novo caractere foi escrito no *buffer* de recepção antes da leitura do que já estava armazenado previamente):

0 - sem erros;

1 - erro de sobrescrita (símbolo **OE**).

BRK - sinalizador de detecção de parada. Uma condição de parada é definida como um período de dez ou mais tempos de *bit*, com a linha em nível baixo:

0 - nenhuma condição de parada;

1 - ocorreu uma condição de parada (símbolo **BRK**).

URXEIE - habilitação da interrupção de recepção para caracteres com erros:

0 - o *flag* URXIFGx não é setado na recepção de caracteres com erros;

1 - o *flag* URXIFGx é setado na recepção de caracteres com erros (símbolo **URXEIE**).

URXWIE - habilitação da interrupção de recepção de caracteres de endereço (caso os caracteres contenham erros e URXEIE = 0, o *flag* não é setado):

0 - todos os caracteres recebidos, setam a interrupção URXIFGx;

1 - somente caracteres de endereço setam o URXIFGx (símbolo **URXWIE**).

RXWAKE - sinalizador do tipo de caractere recebido:

0 - dado;

1 - endereço (símbolo **RXWAKE**).

RXERR - sinalizador de erro de recepção. É setado quando um dos *flags* de erro (FE, PE, OE ou BRK) é também setado. Esse *flag* pode ser apagado simplesmente lendo o conteúdo do registrador de recepção UxRXBUF:

0 - nenhum erro detectado;

1 - caractere(s) recebido(s) com erro(s) (símbolo **RXERR**).

5.8.9.4. UxMCTL

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0073	U0MCTL	Leitura	m7	m6	m5	m4	m3	m2	m1	m0
0x007B	U1MCTL	Escrita								
		Reset	?	?	?	?	?	?	?	?

mx - *bits* de controle do modulador do gerador de *clock* da USART (BRG);

5.8.9.5. UxBR0

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0074	U0BR0	Leitura	8 *bits* menos significativos do divisor de *clock* do BRG							
0x007C	U1BR0	Escrita								
		Reset	?	?	?	?	?	?	?	?

5.8.9.6. UxBR1

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0075 0x007D	U0BR1 U1BR1	Leitura Escrita	8 *bits* mais significativos do divisor de *clock* do BRG							
		Reset	?	?	?	?	?	?	?	?

5.8.9.7. UxRXBUF

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0076 0x007E	U0RXBUF U1RXBUF	Leitura	8 *bits* do último caractere recebido. A leitura desse registrador faz com que os *flags* RXERR, RXWAKE e URXIFGx sejam apagados. No modo de caractere de 7 *bits*, o MSB é sempre igual a zero.							
		Escrita	-							
		Reset	?	?	?	?	?	?	?	?

5.8.9.8. UxTXBUF

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0077 0x007F	U0TXBUF U1TXBUF	Leitura Escrita	*Buffer* de transmissão: contém o caractere a ser transmitido. A escrita nesse registrador apaga o *flag* UTXIFGx. Para caracteres de 7 *bits* o MSB não é utilizado e é descartado.							
		Reset	?	?	?	?	?	?	?	?

5.8.9.9. ME1

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0004	ME1	Leitura Escrita	UTXE0	URXE0						
		Reset	0	0						

* Nos dispositivos 12xx, utilizam-se os *bits* 0 e 1 do ME2 para o controle da USART 0.

UTXE0 - habilitação do transmissor da USART0:
0 - desligado;
1 - ligado (símbolo **UTXE0**).

URXE0 - habilitação do receptor da USART0:
0 - desligado;
1 - ligado (símbolo **URXE0**).

5.8.9.10. ME2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0005	ME2	Leitura			UTXE1	URXE1			UTXE0 *	URXE0 *
		Escrita								
		Reset			0	0			0	0

* Somente nos dispositivos 12xx.

UTXE1 - habilitação da etapa de transmissão da USART1:
0 - desabilitada;
1 - habilitada (símbolo **UTXE1**).

URXE1 - habilitação da etapa de recepção da USART1:
0 - desabilitada;
1 - habilitada (símbolo **URXE1**).

UTXE0 - habilitação da etapa de transmissão da USART0 (somente nos dispositivos 12xx):
0 - desabilitada;
1 - habilitada (símbolo **UTXE0**).

URXE0 - habilitação da etapa de recepção da USART0 (somente nos dispositivos 12xx):
0 - desabilitada;
1 - habilitada (símbolo **URXE0**).

5.8.9.11. IE1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0000	IE1	Leitura	**UTXIE0**	**URXIE0**	ACCVIE	NMIIE	-	-	OFIE	WDTIE
		Escrita								
		Reset	0	0	0	0	0	0	0	0

UTXIE0 - habilitação da interrupção por transmissão da USART0.

URXIE0 - habilitação da interrupção por recepção da USART0.

5.8.9.12. IE2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0001	IE2	Leitura	BTIE	-	UTXIE1	URXIE1	-	-	UTXIE0*	URXIE0*
		Escrita								
		Reset	0	0	0	0	0	0	0	0

* Somente nos dispositivos 12xx.

UTXIE1 - habilitação da interrupção de transmissão da USART1 (símbolo **UTXIE1**).

URXIE1 - habilitação da interrupção de recepção da USART1 (símbolo **URXIE1**).

UTXIE0 - habilitação da interrupção por transmissão da USART0 (somente nos modelos 12xx) (símbolo **UTXIE0**).

URXIE0 - habilitação da interrupção por recepção da USART0 USART0 (somente nos modelos 12xx) (símbolo **URXIE0**).

5.8.9.13. IFG1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0002	IFG1	Leitura / Escrita	UTXIFG0	URXIFG0	-	NMIIFG	RSTIFG	PORIFG	OFIFG	WDTIFG
		Reset	1	0	0	0	0	0	0	0

UTXIFG0 - sinalizador da interrupção de transmissão da USART0.
URXIFG0 - sinalizador da interrupção de recepção da USART0.

5.8.9.14. IFG2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0003	IFG2	Leitura / Escrita	BTIFG	-	UTXIFG1	URXIFG1	-	-	UTXIFG0 *	URXIFG0 *
		Reset	0	0	1	0	0	0	1	0

* Somente nos dispositivos 12xx.

UTXIFG1 - sinalizador da interrupção de transmissão da USART1 (símbolo **UTXIFG1**).
URXIFG1 - sinalizador da interrupção de recepção da USART1 (símbolo **URXIFG1**).
UTXIFG0 - sinalizador da interrupção de transmissão da USART0 (somente nos modelos 12xx) (símbolo **UTXIFG0**).
URXIFG0 - sinalizador da interrupção de recepção da USART0 (somente nos modelos 12xx) (símbolo **URXIFG0**).

5.8.10. Exemplos de Utilização

O exemplo seguinte demonstra a configuração da USART para operar em uma velocidade de 9.600 bps, 8N1 (8 *bits* de dados, sem paridade, 1 *stop bit*). O programa utiliza o DCO como fonte de *clock* (freqüência aproximada de 823KHz). Cada caractere recebido é retornado ao terminal.

```
#include <io430x14x.h>
//*************************************************************
// Ecoando todos os caracteres recebidos
// Velocidade = 9600 Bps, 8N1
// Clock = DCO ~= 823KHz
//*************************************************************

void main( void )
{
   WDTCTL = WDTPW + WDTHOLD;     // desativa o watchdog
   // Configura os pinos da USART
   P3SEL_bit.P3SEL_4 = 1;
   P3SEL_bit.P3SEL_5 = 1;
   // Configura a USART
   ME1 = URXE0 + UTXE0;  // habilita o módulo
   U0CTL = SWRST + CHAR; // caractere de 8 bits
   U0TCTL = SSEL1 + SSEL0;       // clock da USART = SMCLK
   U0MCTL = 0xDD;
   U0BR0 = 0x55;
   U0BR1 = 0x00;
   U0CTL_bit.SWRST = 0;  // apaga o reset da USART
```

```
    while (1)
    {
       // verifica se chegou um novo caractere
       if (IFG1_bit.URXIFG0)
       {
          // escreve o caractere recebido no buffer de transmissão
          U0TXBUF = U0RXBUF;
          // aguarda a transmissão ser completada
          while (!U0TCTL_bit.TXEPT);
       }
    }
}
```

Exemplo 5-9

Para propósitos de teste, podemos modificar a linha: `U0TXBUF = U0RXBUF` para: `U0TXBUF = U0RXBUF+1`, de forma que o programa retorna sempre o caractere imediatamente posterior (na tabela ASCII) ao recebido.

No programa seguinte, utilizamos a interrupção de recepção de dados (IFG1:URXIFG0) para retornar o caractere para o terminal. Cada caractere recebido dispara uma interrupção. A RTI lê o caractere recebido e o envia de volta ao terminal. Novamente a velocidade escolhida é de 9.600 bps, 8N1 com *clock* interno de 823KHz (DCO):

```
#include <io430x14x.h>
#include <intrinsics.h>

//*************************************************************
// Ecoando todos os caracteres recebidos, utilizando a inter-
// rupção
// Velocidade = 9600 Bps, 8N1
// Clock = DCO ~= 823KHz
//*************************************************************

#pragma vector = USART0RX_VECTOR
__interrupt void trata_usartrx0(void)
{
   // Apaga erros
   U0RCTL = 0;
   // Lê o dado e envia
   U0TXBUF = U0RXBUF;
}
void main( void )
{
   WDTCTL = WDTPW + WDTHOLD;     // desativa o watchdog
   // Configura os pinos da USART
   P3SEL_bit.P3SEL_4 = 1;
   P3SEL_bit.P3SEL_5 = 1;
   // Configura a USART
   ME1 = URXE0 + UTXE0;  // habilita o módulo
   U0CTL = SWRST + CHAR; // caractere de 8 bits
   U0TCTL = SSEL1 + SSEL0;       // clock da USART = SMCLK
   U0MCTL = 0xDD;
   U0BR0 = 0x55;
   U0BR1 = 0x00;
   U0CTL_bit.SWRST = 0;  // apaga o reset da USART
   // Habilita a interrupção de recepção da USART0
   IE1_bit.URXIE0 = 1;
   // Habilita as interrupções
   __enable_interrupt();
   while (1);
}
```

Exemplo 5-10

5.9. USART - Modo Síncrono SPI

A USART dos MSP430 pode também operar no modo síncrono. Neste caso, o protocolo utilizado é compatível com o SPI (*Serial Peripheral Interface*). Esse tipo de interface pode ser utilizado para comunicação com conversores A/D e D/A externos, memórias e cartões de memória FLASH, potenciômetros digitais, etc.

O diagrama básico da USART, quando opera nesse modo, pode ser visto na figura 5-23. Ele é bastante semelhante ao da USART quando opera no modo assíncrono.

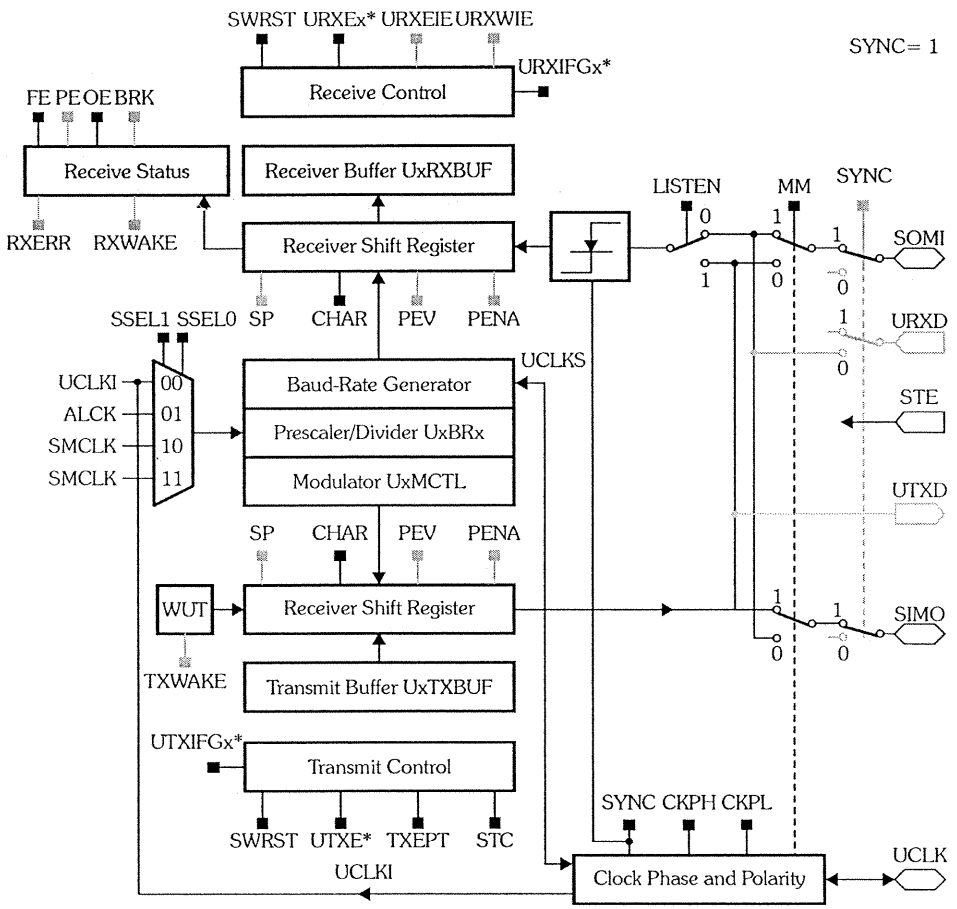


Figura 5-23

A seleção do modo SPI é feita pelo *bit* UxCTL:SYNC. Com ele em nível "1", selecionamos o modo SPI (desde que o *bit* UxCTL:I2C esteja apagado, caso ele esteja disponível no *chip* utilizado).

O SPI é um protocolo síncrono do tipo mestre-escravo, ou seja, a comunicação é comandada e iniciada por um dispositivo (o mestre) e no outro lado, um dispositivo (escravo) responde às solicitações do mestre. Existem duas linhas para a troca de dados e outra linha para o sinal de *clock*. As linhas possuem a seguinte nomenclatura:

- SIMO (*Slave In Master Out*), MOSI ou SI - para o envio de informações (modo mestre) ou recepção (modo escravo);
- SOMI (*Slave Out Master In*), MISO ou SO - para o envio de informações (modo escravo) ou recepção (modo mestre);
- UCLK, *Clock* ou SCLK - envio de *clock* (modo mestre) ou recepção de *clock* (modo escravo).

Além desses sinais, um quarto normalmente está presente: o sinal de seleção de *chip* (STE, CS ou SS) que permite que um dispositivo mestre selecione um entre vários dispositivos escravos. Além disso, esse sinal é responsável por indicar ao dispositivo escravo o início e o término da comunicação.

Em muitas aplicações, as linhas SIMO e SOMI podem ser interligadas, de forma a minimizar a quantidade de conexões entre o dispositivo mestre e o(s) escravo(s).

O protocolo especifica ainda quatro modos de funcionamento possíveis. A diferença entre eles diz respeito à fase do sinal de *clock* em que o dado é lido. Nos modos 0 e 1, o sinal de *clock* é ativo em nível alto e nos modos 2 e 3 o *clock* é ativo em nível baixo. A seleção entre um modo e outro é feita pelos *bits* UxTCTL:CKPH (fase do sinal de *clock*) e UxTCTL:CKPL (polaridade do sinal de *clock*), conforme a tabela seguinte:

Modo	CKPL	CKPH	Descrição
0	0	0	O dado é armazenado na borda de subida do *clock*
1	0	1	O dado é armazenado na borda de descida do *clock*
2	1	0	O dado é armazenado na borda de descida do *clock*
3	1	1	O dado é armazenado na borda de subida do *clock*

Tabela 5-20

A escolha de um ou outro modo vai depender do dispositivo com o qual deseja se comunicar.

A figura 5-24 representa e relaciona os diversos sinais envolvidos em uma comunicação SPI. Observe a relação entre o sinal de *clock* (UCLK) e os sinais de dados nos quatro modos de operação do SPI.

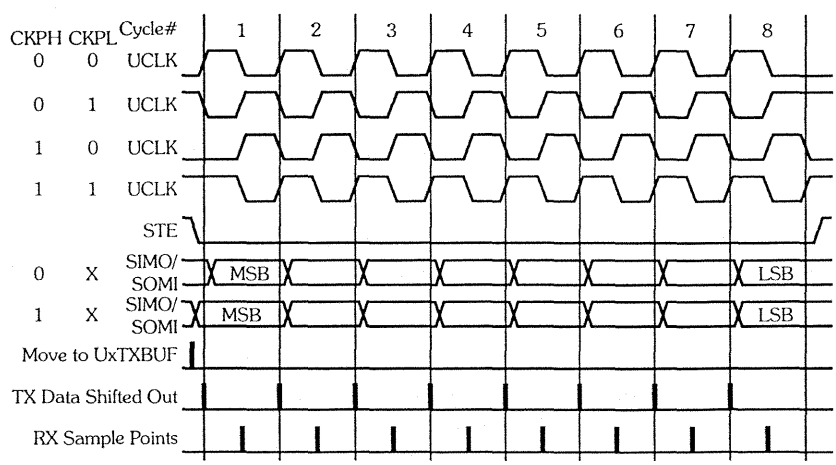


Figura 5-24

5.9.1. Gerador de *Clock*

O circuito gerador de *baud-rate* da USART é o responsável pela geração do sinal de *clock* para a comunicação SPI no modo mestre.

A principal diferença em relação ao modo assíncrono é que, nesse modo, o modulador do BRG não é utilizado, sendo recomendável manter o registrador UxMCTL com o valor zero.

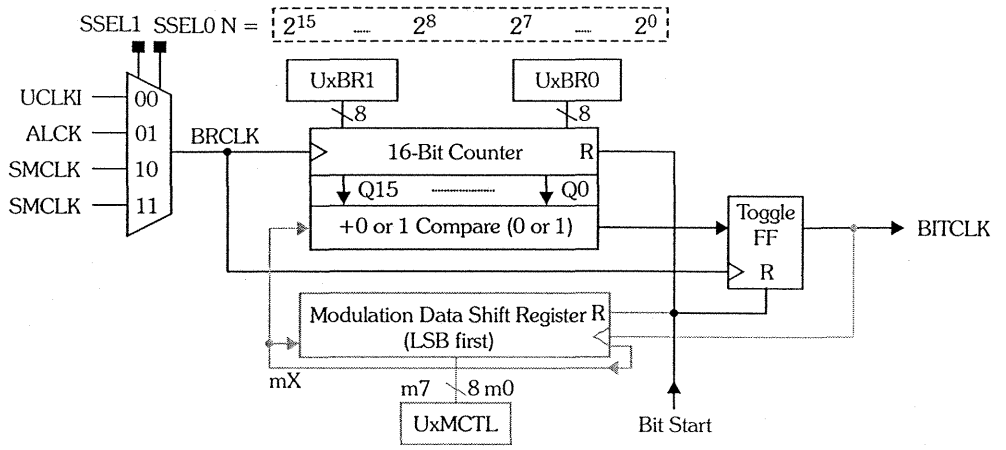


Figura 5-25

A velocidade de comunicação da interface SPI será:

$$Velocidade_{SPI} = BRCLK / UxBR$$

sendo:

- BRCLK é a freqüência da fonte de *clock* da USART;
- UxBR é o valor de 16 *bits* armazenado em UxBR1 e UxBR0.

O fabricante recomenda que não seja utilizado um fator de divisão do *clock* menor que 2 (ou seja, UxBR deve ser maior ou igual a 2). Valores menores que este podem provocar o funcionamento incorreto da USART.

5.9.2. Configuração da USART

Para operação no modo SPI, os seguintes passos devem ser seguidos:

1. Configurar os pinos da USART para a função alternativa (veja a tabela 5-21).
2. Ativar o *reset* da USART (*bit* SWRST do registrador UxCTL).
3. Configurar os registradores da USART:
   - UxCTL: SYNC = 1, I2C = 0, SWRST = 0 e os demais *bits* conforme o modo desejado;
   - UxTCTL: fonte de *clock* (*bits* SSELx), polaridade (*bit* CKPL) e fase (*bit* CKPH), além da habilitação do pino STE (*bit* STC);
   - Velocidade de comunicação (registradores UxBR0, UxBR1). O registrador UxMCTL deve ser mantido em zero.

4. Habilitar a USART (*bit* USPIEx, nos registradores ME1 ou ME2, conforme a USART desejada).
5. Desativar o *reset* da USART (*bit* SWRST = 0).
6. Caso deseje utilizar as interrupções da USART, habilitá-las por meio dos *bits* UTXIEx e/ou URXIEx (registradores IE1 ou IE2, conforme a USART desejada). **Lembre-se: as interrupções somente podem ser ativadas com SWRST=0 !**

   Lembre-se de que a ativação do sinal de *reset* da USART (UxCTL:SWRST =1) faz com que os *bits* URXIEx, UTXIEx, URXIFGx, OE e FE sejam apagados e o *bit* UTXIFGx seja setado.

5.9.3. Operação no Modo Mestre

Para a operação no modo mestre, é necessário configurar o *bit* UxCTL:MM = 1.

Nesse modo, o pino SIMO é configurado como saída, SOMI como entrada e UCLK como saída. Também é importante observar se o pino STE será utilizado: nessa modalidade de operação, se esse pino estiver ativado, é utilizado para previnir a transmissão simultânea de dois dispositivos mestres (evento tecnicamente chamado de colisão). Caso o pino STE esteja em nível "1", a transmissão é feita normalmente. Caso o pino esteja em nível "0", os pinos SIMO e UCLK são configurados como entradas, de forma a evitar que interfiram no barramento e o *flag* FE é setado, indicando o estado de erro.

Caso não deseje utilizar esse pino, é importante configurar o *bit* UxTCTL:STC = 1.

Para realizar uma transmissão, basta escrever um dado no registrador UxTXBUF, que provoca a carga desse dado no registrador de deslocamento de transmissão e seta o *flag* IFGx:UTXIFGx. Durante a transmissão, o *bit* UxTCTL:TXEPT permanece em nível "0". Após a transmissão ser completada, esse *bit* é setado.

Como nessa modalidade o sinal de *clock* é fornecido pelo mestre, para receber um dado é necessário escrever um valor qualquer no registrador UxTXBUF, o que provoca o início da transmissão e também da recepção do caractere. O *flag* IFGx:URXIFGx é setado após a recepção do caractere ter sido completada.

Observe que, quando operando no modo de sete *bits* (UxCTL:CHAR = 0), a informação a ser transmitida deve ser alinhada pelo MSB (justificada à esquerda) antes de ser escrita no UxTXBUF, com o LSB desprezado. Já a informação recebida no UxRXBUF é alinhada à direita (o MSB é sempre "0").

5.9.4. Operação no Modo Escravo

No modo escravo (UxCTL:MM = 0), a USART necessita da um sinal de *clock* externo (proveniente do mestre), por isso o pino UCLK, neste modo, é configurado como entrada, o pino SIMO é configurado como entrada e o pino SOMI é configurado como saída.

A operação de transmissão é feita pela escrita no registrador UxTXBUF, no entanto ela somente tem início quando o dispositivo mestre realiza uma transmissão para o escravo. Neste caso, o dado presente no UxTXBUF é carregado no registrador de deslocamento de transmissão e o *flag* IFGx:UTXIFGx é setado. Após o término da transmissão, o *bit* UxTCTL:TXEPT é setado, indicando que a transmissão foi completada. Ao mesmo tempo em que o dado presente no *buffer*

de transmissão é transmitido para o mestre, a informação proveniente dele é recebida no UxRXBUF. Após o término da recepção, o *flag* IFGx:URXIFGx é setado, indicando esta situação.

Repare que, caso o pino STE seja utilizado (UxTCTL:STC = 1), o dispositivo escravo somente opera quando o pino estiver em nível lógico "0". Caso o pino esteja em nível "1", qualquer operação de recepção no pino SIMO é paralisada e o pino SOMI é configurado como entrada (de forma a não interferir em outros dispositivos escravos).

5.9.5. Interrupções

A USART possui duas fontes de interrupção distintas: a transmissão de caractere (*flag* IFGx:UTXIFGx) e a recepção de caractere (*flag* IFGx:URXIFGx).

O *flag* UTXIFGx é setado sempre que o dado presente no *buffer* de transmissão tenha sido completamente transferido para o registrador de deslocamento do transmissor. O apagamento desse *flag* pode ser feito de uma das seguintes formas:

1. Pela chamada automática da rotina de tratamento dessa interrupção;
2. Pela escrita de um novo caractere no registrador UTXBUF;
3. Pela escrita do valor "0" no *flag*.

Observe que, após um *reset* do sistema (PUC) ou da USART (UxCTL:SWRST = 1), o *flag* IFGx:UTXIFGx é automaticamente setado (indicando que a USART está pronta para receber um novo caractere) e a interrupção desabilitada (*bit* IEx:UTXIEx = 0).

O *flag* URXIFGx é setado após a recepção de um caractere e o seu apagamento pode ocorrer de uma das seguintes formas:

1. Pelo desvio automático para a rotina de tratamento da interrupção de recepção de caractere;
2. Pela leitura do *buffer* de recepção (UxRXBUF);
3. Pela escrita do valor "0" no *flag*.

5.9.6. Conexões da USART

Os sinais da(s) USART(s) estão disponíveis nos seguintes pinos:

Sinal	Pino
UCLK0	P3.3 (família 1xx e 4xx 80 e 100 pinos) P2.1 (família 4xx 64 pinos)
SIMO0	P3.1 (família 1xx e 4xx 80 e 100 pinos) P1.6 (família 4xx 64 pinos)
SOMI0	P3.2 (família 1xx e 4xx 80 e 100 pinos) P1.7 (família 4xx 64 pinos)
STE0	P3.0 (família 1xx e 4xx 80 e 100 pinos) P2.2 (família 4xx 64 pinos)

Sinal	Pino
UCLK1	P5.3 (1xx) P4.5 (44x)
SIMO1	P5.1 (1xx) P4.3 (44x)
SOMI1	P5.2 (1xx) P4.4 (44x)
STE1	P5.0 (1xx) P4.2 (44x)

Tabela 5-21

5.9.7. Registradores da USART no Modo SPI

Quando opera no modo SPI, a USART utiliza os seguintes registradores para o seu funcionamento:

- **UxCTL** - para configuração geral da USART;
- **UxTCTL** - para configuração do transmissor da USART;
- **UxRCTL** - para configuração do receptor da USART;
- **UxMCTL, UxBR0 e UxBR1** - para configuração do gerador de *clock*;
- **UxRXBUF** - para armazenamento do último caractere recebido;
- **UxTXBUF** - para armazenamento do caractere a ser transmitido;
- **MEx** - para habilitação da USART;
- **IEx** - para habilitação das interrupções da USART;
- **IFGx** - para sinalização das interrupções da USART;
- **PxSEL** - seleção da função alternativa para os pinos da USART.

5.9.7.1. UxCTL

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0070 0x0078	U0CTL U1CTL	Leitura Escrita	Não Utilizado	Não Utilizado	I2C*	CHAR	LISTEN	SYNC	MM	SWRST
		Reset	0	0	0	0	0	0	0	0

I2C - ativação do modo I^2C (somente na USART0 dos modelos 15x e 16x):
0 - modo SPI;
1 - modo I^2C (símbolo **I2C**).

CHAR - largura do caractere (7 ou 8 *bits*):
0 - 7 *bits*;
1 - 8 *bits* (símbolo **CHAR**).

LISTEN - habilitação do *loopback* (a transmissão e a recepção são internamente conectadas):
0 - modo normal;
1 - modo *loopback*, os sinais de saída e de entrada (SIMO e SOMI) são interconectados internamente. Todos os dados transmitidos são recebidos pela etapa de recepção (símbolo **LISTEN**).

SYNC - seleção do modo síncrono:
0 - modo assíncrono;
1 - modo síncrono (SPI ou I^2C) (símbolo **SYNC**).

MM - seleção do modo mestre:
0 - a USART é um dispositivo escravo;
1 - a USART é um dispositivo mestre (símbolo **MM**).

SWRST - ativação do *reset* da USART:
0 - USART no modo normal;
1 - USART no modo *reset* (símbolo **SWRST**).

5.9.7.2. UxTCTL

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0071 0x0079	U0TCTL U1TCTL	Leitura Escrita	CKPH	CKPL	SSELx		Não utilizado	Não utilizado	STC	TXEPT
		Reset	0	0	0	0	0	0	0	1

CKPH - seleção da fase do sinal de *clock* para modo síncrono:
0 - o *clock* interno (UCLKI) é igual ao *clock* externo (UCLK);
1 - o *clock* interno (UCLKI) é igual ao *clock* externo (UCLK) invertido (símbolo **CKPH**).

CKPL - seleção da polaridade de *clock* para modo síncrono:
0 - o *clock* interno (UCLKI) é igual ao *clock* externo (UCLK);
1 - o *clock* interno (UCLKI) é igual ao *clock* externo (UCLK) invertido (símbolo **CKPL**).

SSELx - Seleção da fonte de *clock* para o gerador de *baud-rate* (símbolos **SSEL1** e **SSEL0**):
00 - UCLKI;
01 - ACLK;
10 - SMCLK;
11 – SMCLK.

STC - controle de habilitação do pino STE:
0 - pino STE habilitado;
1 - pino STE desabilitado (símbolo **STC**).

TXEPT - sinalizador de término de transmissão:
0 - há informação sendo transmitida ou aguardando no *buffer* UxTXBUF;
1 - não há informação sendo transmitida e o *buffer* UxTXBUF está vazio (ou a USART está em modo de *reset*, com SWRST = 1) (símbolo **TXEPT**).

5.9.7.3. UxRCTL

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0072 0x007A	U0RCTL U1RCTL	Leitura Escrita	FE	Não utilizado	OE	Não utilizado	Não utilizado	Não utilizado	Não utilizado	Não utilizado
		Reset	0	0	0	0	0	0	0	0

FE - sinalizador de erro de quadro (sinaliza uma colisão de barramento quando o pino STE está habilitado no modo mestre):
0 - sem erros;
1 - uma borda de descida foi detectada no pino STE (símbolo **FE**).

OE - sinalizador de erro de sobrescrita (indica que um novo caractere foi escrito no *buffer* de recepção antes da leitura do que já estava armazenado previamente). Esse *bit* é automaticamente apagado pela leitura do UxRXBUF, por um *reset* (PUC ou SWRST), ou ainda pela escrita do valor "0" nele:
0 - sem erros;
1 - erro de sobrescrita (símbolo **OE**).

5.9.7.4. UxMCTL

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0073 0x007B	U0MCTL U1MCTL	Leitura Escrita	m7	m6	m5	m4	m3	m2	m1	m0
		Reset	?	?	?	?	?	?	?	?

mx - *bits* de controle do modulador do gerador de *clock* da USART (BRG) devem ser mantidos em "0" no modo SPI.

5.9.7.5. UxBR0

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0074 0x007C	U0BR0 U1BR0	Leitura Escrita	8 *bits* menos significativos do divisor de clock do BRG							
		Reset	?	?	?	?	?	?	?	?

5.9.7.6. UxBR1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0075 0x007D	U0BR1 U1BR1	Leitura Escrita	8 *bits* mais significativos do divisor de clock do BRG							
		Reset	?	?	?	?	?	?	?	?

5.9.7.7. UxRXBUF

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0076 0x007E	U0RXBUF U1RXBUF	Leitura	8 *bits* do último caractere recebido. A leitura desse registrador faz com que os *flags* OE e URXIFGx sejam apagados. No modo de caractere de 7 *bits*, o MSB é sempre igual a zero.							
		Escrita	-							
		Reset	?	?	?	?	?	?	?	?

5.9.7.8. UxTXBUF

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0077 0x007F	U0TXBUF U1TXBUF	Leitura Escrita	Buffer de transmissão: contém o caractere a ser transmitido. A escrita nesse registrador apaga o *flag* UTXIFGx. Para caracteres de 7 *bits* o LSB não é utilizado.							
		Reset	?	?	?	?	?	?	?	?

5.9.7.9. ME1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0004	ME1	Leitura		**USPIE0**						
		Escrita								
		Reset		0						

USPIE0 - habilitação do módulo SPI0 (TX e RX):
0 - desligado;
1 - ligado (símbolo **USPIE0**).

5.9.7.10. ME2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0005	ME2	Leitura				**USPIE1**				**USPIE0***
		Escrita								
		Reset				0				0

* Somente nos dispositivos 12xx.

USPIE1 - habilitação do módulo SPI1 (TX e RX):
0 - desligado;
1 - ligado (símbolo **USPIE1**).

USPIE0 - habilitação do módulo SPI0 (somente nos modelos 12xx) (TX e RX):
0 - desligado;
1 - ligado (símbolo **USPIE0**).

5.9.7.11. IE1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0000	IE1	Leitura	**UTXIE0**	**URXIE0**	ACCVIE	NMIIE	-	-	OFIE	WDTIE
		Escrita								
		Reset	0	0	0	0	0	0	0	0

UTXIE0 - habilitação da interrupção por transmissão da USART0 (símbolo **UTXIE0**).
URXIE0 - habilitação da interrupção por recepção da USART0 (símbolo **URXIE0**).

5.9.7.12. IE2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0001	IE2	Leitura	BTIE	-	**UTXIE1**	**URXIE1**	-	-	**UTXIE0***	**URXIE0***
		Escrita								
		Reset	0	0	0	0	0	0	0	0

* Somente nos dispositivos 12xx.

UTXIE1 - habilitação da interrupção de transmissão da USART1 (símbolo **UTXIE1**).

URXIE1 - habilitação da interrupção de recepção da USART1 (símbolo **URXIE1**).

UTXIE0 - habilitação da interrupção por transmissão da USART0 (somente nos modelos 12xx) (símbolo **UTXIE0**).

URXIE0 - habilitação da interrupção por recepção da USART0 (somente nos modelos 12xx) (símbolo **URXIE0**).

5.9.7.13. IFG1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0002	IFG1	Leitura / Escrita	UTXIFG0	URXIFG0	-	NMIIFG	RSTIFG	PORIFG	OFIFG	WDTIFG
		Reset	1	0	0	0	0	0	0	0

UTXIFG0 - sinalizador da interrupção de transmissão da USART0.

URXIFG0 - sinalizador da interrupção de recepção da USART0.

5.9.7.14. IFG2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0003	IFG2	Leitura / Escrita	BTIFG	-	UTXIFG1	URXIFG1	-	-	UTXIFG0 *	URXIFG0 *
		Reset	0	0	1	0	0	0	1	0

* Somente nos dispositivos 12xx.

UTXIFG1 - sinalizador da interrupção de transmissão da USART1 (símbolo **UTXIFG1**).

URXIFG1 - sinalizador da interrupção de recepção da USART1 (símbolo **URXIFG1**).

UTXIFG0 - sinalizador da interrupção de transmissão da USART0 (somente nos modelos 12xx) (símbolo **UTXIFG0**).

URXIFG0 - sinalizador da interrupção de recepção da USART0 (somente nos modelos 12xx) (símbolo **URXIFG0**).

5.9.8. Exemplos de Utilização

As funções seguintes foram escritas para efetuar a comunicação entre um MSP430F149 e o *chip* HT1381 da Holtek, um relógio de tempo real com calendário e comunicação SPI por dois fios (um para o *clock* e outro bidirecional para os dados).

Repare que as funções não utilizam o pino STE. A função de controle de habilitação do dispositivo escravo (o RTC) é realizada por um pino de E/S qualquer, como veremos no tópico 7.7, que apresenta essa operação em maiores detalhes.

```
//*********************************************************************
// Inicializa a USART para o modo SPI
//*********************************************************************
void usart_spi_inicializa()
{
   // Configura os pinos da USART
   P3SEL_bit.P3SEL_1 = 1;
   P3SEL_bit.P3SEL_2 = 1;
   P3SEL_bit.P3SEL_3 = 1;
   // Configura a USART
   ME1 = USPIE0;  // habilita o módulo
   // Modo = Síncrono, Master, 8 Bits
   U0CTL = SWRST + CHAR + SYNC + MM;
   // Clock = SMCLK, STE desabilitado, polaridade 1
   U0TCTL = SSEL1 + SSEL0 + STC + CKPH;
   U0MCTL = 0;    // modulador = 0
   U0BR0 = 0x2;   // UCLK = SMCLK / 2
   U0BR1 = 0x00;
   U0CTL_bit.SWRST = 0;  // tira a USART do reset
}
//*********************************************************************
// Escreve um comando seguido de um dado de 8 bits na SPI
//*********************************************************************
void usart_spi_escreve(unsigned char comando, unsigned char dado)
{
   unsigned char temp;
   // Configura o pino P3.1 para a função SPI (SIMO)
   P3SEL_bit.P3SEL_1 = 1;
   // Transmite o comando
   U0TXBUF = comando;
   // Aguarda a transmissão
   while (!U0TCTL_bit.TXEPT);
   // Lê o caractere recebido
   temp = U0RXBUF;
   // Transmite o dado
   U0TXBUF = dado;
   // Aguarda a transmissão
   while (!U0TCTL_bit.TXEPT);
   // Lê o caractere recebido
   temp = U0RXBUF;
}
//*********************************************************************
// Escreve um comando de 8 bits e recebe um dado de 8 bits da SPI
//*********************************************************************
unsigned char usart_spi_le(unsigned char comando)
{
   unsigned char temp;
   // Configura o pino P3.1 para a função SPI (SIMO)
   P3SEL_bit.P3SEL_1 = 1;
   // Transmite o comando
   U0TXBUF = comando;
   // Aguarda a transmissão
   while (!U0TCTL_bit.TXEPT);
   // Lê o caractere recebido
   temp = U0RXBUF;
   // Configura o pino P3.1 como entrada
   P3SEL_bit.P3SEL_1 = 0;
   P3DIR_bit.P3DIR_1 = 0;
   // Envia
   U0TXBUF = comando;
   while (!U0TCTL_bit.TXEPT);
   return(U0RXBUF);
}
```

Exemplo 5-11

5.10. USART - Modo Síncrono I^2C

A terceira modalidade de funcionamento da USART, disponível apenas nos modelos 15x e 16x, é o I^2C.

O I^2C (*Inter Integrated Comunication* ou Comunicação entre Integrados) é um protocolo síncrono *half duplex* a dois fios do tipo mestre-escravo, criado pela Philips para facilitar o desenvolvimento de sistemas modulares para televisores e outros aparelhos eletrônicos de consumo geral.

5.10.1. Características do Protocolo

As duas linhas utilizadas são a de *clock* (chamada de SCL) e outra de dados (chamada de SDA). Graças à especificação de saídas em coletor (ou dreno) aberto, o protocolo permite a ligação de diversos dispositivos nas mesmas linhas, formando um autêntico barramento de comunicação serial, ou uma rede de dispositivos.

Atualmente o protocolo está em sua revisão número 2.1 e suporta velocidades de até 3,4*Mbps*. No entanto, a grande maioria dos dispositivos I^2C é compatível com as versões anteriores do protocolo, e portanto limitadas a velocidades de 100 ou 400 Kbps.

A quantidade de dispositivos presentes no barramento é apenas limitada pela capacitância máxima admitida que é de 400pF.

Na figura 5-26 temos um exemplo de conexão de circuitos integrados a um barramento I^2C. Repare na existência dos resistores de *pull-up* "Rp", utilizados para manter as linhas do barramento em nível lógico "1".

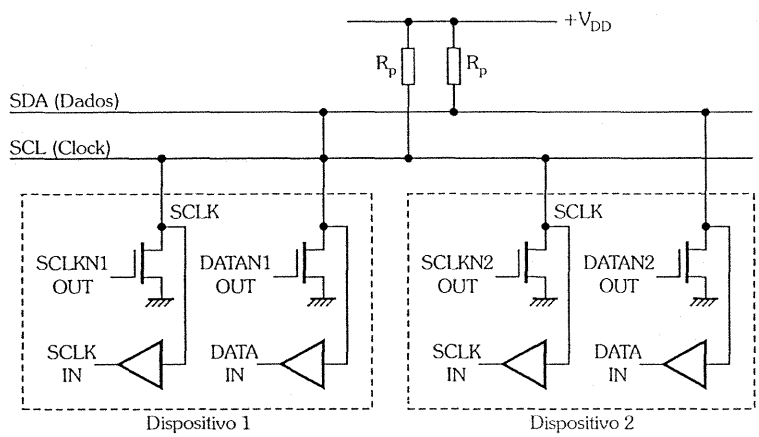


Figura 5-26

O I^2C suporta também o chamado *multimastering*, ou seja, a presença de diversos mestres simultaneamente no mesmo barramento. No entanto, durante uma comunicação, somente um dos mestres pode estar ativo, ou ocorrerá uma colisão no barramento.

O funcionamento do protocolo baseia-se em alguns princípios básicos:

1. A informação presente na linha de dados (SDA) somente é lida durante a fase alta da linha de *clock* (SCL).

2. Somente é permitido alterar o nível da linha de dados (SDA) durante a fase baixa da linha de *clock* (SCL).
3. Quando o barramento não está em uso, ambas as linhas permanecem desligadas (e, portanto forçadas em nível "1" pelos resistores de *pull-up*).

Para sinalizar o início e o fim de uma transmissão, utiliza-se a violação da segunda regra anterior.

Assim, para sinalizar o início da transmissão (também chamado de condição de partida ou *START*), o dispositivo força a linha SDA de "1" para "0", durante a fase alta do *clock*. Essa violação indica aos dispositivos que uma transmissão terá início.

Para sinalizar o fim de uma transmissão, utiliza-se a chamada condição de parada, ou *STOP*, que consiste na transição de "0" para "1" da linha SDA durante a fase alta da linha SCL. Note que a condição de parada somente é emitida ao término da comunicação e não ao término de um caractere.

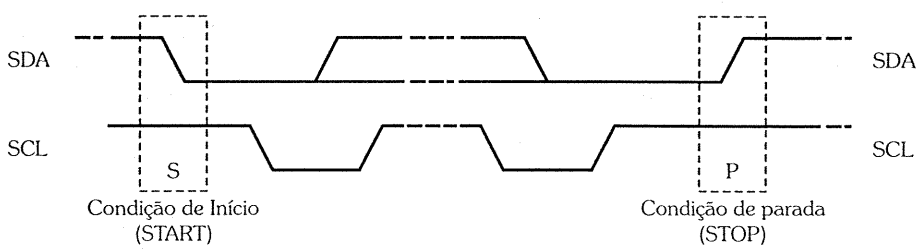


Figura 5-27

Após o *bit* de *START*, são transmitidos oito *bits* de dados, iniciando pelo MSB. Após o último *bit* (LSB) o receptor deve gerar uma condição de reconhecimento (ACK - *acknowledge*), o que é feito forçando a linha SDA em nível "0" antes do nono pulso de *clock* da linha SCL.

Caso o receptor não reconheça o dado (mantendo a linha SDA em "1" durante o nono pulso da linha SCL), o transmissor deve abortar (gerando uma condição de parada) e reiniciar a transmissão.

5.10.1.1. Formatos de Dados

Para a coexistência de diversos dispositivos em um mesmo barramento, é necessário que cada um possua identificação ou endereço próprio.

O formato básico de um comando I^2C é constituído por 7 *bits* de endereço, utilizados para especificar o dispositivo escravo a ser acessado, seguidos por um *bit* indicador de leitura/escrita, conforme a figura 5-28.

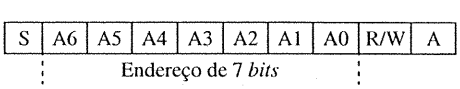


Figura 5-28

Normalmente o endereço de 7 *bits* é composto por duas partes: a primeira, de 4 *bits*, especifica o tipo de dispositivo escravo a ser acessado. A segunda, de 3 *bits*, especifica um entre até oito dispositivos daquele tipo, o qual será acessado.

O *bit* R/W indica se a operação é de leitura (nível "1") ou de escrita (nível "0").

Um detalhe importante é que alguns valores de endereço possuem aplicação predefinida, conforme a tabela 5-22.

Endereço	R/W	Significado
0000000	0	Endereço de chamada geral (*general call*)
0000000	1	*Byte* de partida (pode ser utilizado para auxiliar dispositivos lentos a identificar o início de uma transmissão)
0000001	x	Endereço CBUS
0000010	x	Reservado para formatos diferentes de barramentos
0000011	x	Reservado para propósitos futuros
00001xx	x	Código para modo mestre de alta velocidade (3.4Mbps)
0010xxx	x	Sintetizadores de voz
0011xxx	x	Interfaces de áudio PCM
0100xxx	x	Geradores de tons de áudio
0111xxx	x	*Displays* LED/LCD
1000xxx	x	Interfaces de vídeo
1001xxx	x	Interfaces A/D e D/A
1010xxx	x	Memórias seriais
1100xxx	x	Sintonizadores de RF
1101xxx	x	Relógios / calendários
11111xx	x	Reservado para propósitos futuros
11110xx	x	Modo de endereçamento de 10 *bits*

Tabela 5-22

5.10.1.2. Chamada Geral

O endereço de chamada geral pode ser utilizado para enviar dados ou comandos a todos os dispositivos conectados ao barramento.

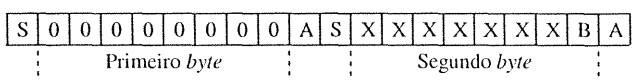


Figura 5-29

O primeiro *byte* do pacote contém o endereço de chamada geral, já o segundo *byte* pode possuir diferentes finalidades, dependendo o estado do *bit* B.

Quando B = 0, podemos ter um dos seguintes comandos:

- 00000110 e 00000100 - escrita da parte programável do endereço escravo. O procedimento realizado deve ser observado na documentação do dispositivo.
- 00000000 - código não permitido para ser utilizado no segundo *byte*.

Os demais códigos (com B=0) não possuem significado e devem ser ignorados pelo dispositivo escravo.

Quando B=1, temos um *hardware general call*, ou chamada geral de *hardware*. Neste caso, os sete *bits* do segundo *byte* contêm o endereço do dispositivo mestre que gerou a mensagem. Essa facilidade pode ser utilizada pelo dispositivo mestre quando ele não conhece o endereço do dispositivo escravo conectado ao barramento. Neste caso, o dispositivo escravo deve ser capaz de responder a esse tipo de mensagem.

5.10.1.3. *Byte* de partida

Quando se utilizam dispositivos muito lentos conectados ao barramento I^2C, pode acontecer de não responderem adequadamente aos comandos do protocolo. Isso acontece principalmente quando se escreve uma interface I^2C por *software* para um microcontrolador atuar como um dispositivo escravo no barramento.

Neste caso, pode acontecer que ele não reconheça suficientemente rápido a condição de partida, o que pode comprometer a comunicação.

Nestes casos, é possível utilizar um recurso adicional previsto no protocolo, que é o *byte* de partida.

Trata-se de uma mensagem especial (00000001) enviada pelo dispositivo mestre e que permite que o dispositivo escravo mais lento reconheça a transmissão e receba corretamente os próximos *bytes* transmitidos.

O formato de uma comunicação típica utilizando um *byte* de partida pode ser visto na figura 5-30.

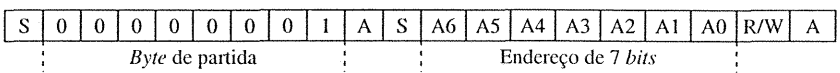


Figura 5-30

5.10.1.4. Modo de Endereçamento de 10 *Bits*

O formato de uma palavra de endereçamento de 10 *bits* é composto de 2 *bytes* conforme em seguida:

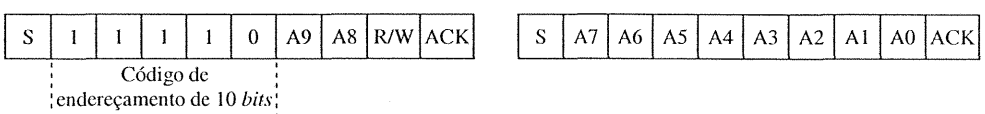


Figura 5-31

sendo:

- S - condição de início;
- A9 a A0 - *bits* de endereçamento;
- R/W - *bit* de escrita/leitura;
- ACK- reconhecimento.

5.10.2. Características do *Hardware*

A interface I^2C implementada nos MSP430 possui as seguintes características:

- Transferências de *byte* e *word*;
- Suporte a endereçamento de 7 e 10 *bits*;
- Chamada geral (*General Call*);
- Suporte a arbitramento do barramento (*multimaster*);
- Velocidades de 100 e 400Kbps;

- *Buffers* de transmissão e de recepção dotados de FIFO;
- Contagem automática dos *bytes* de dados;
- Desenhada para operação em baixa potência com extensas capacidades de interrupção e possibilidade de efetuar a saída de um modo de baixa potência da CPU pela recepção de um *bit* de partida (*START*).

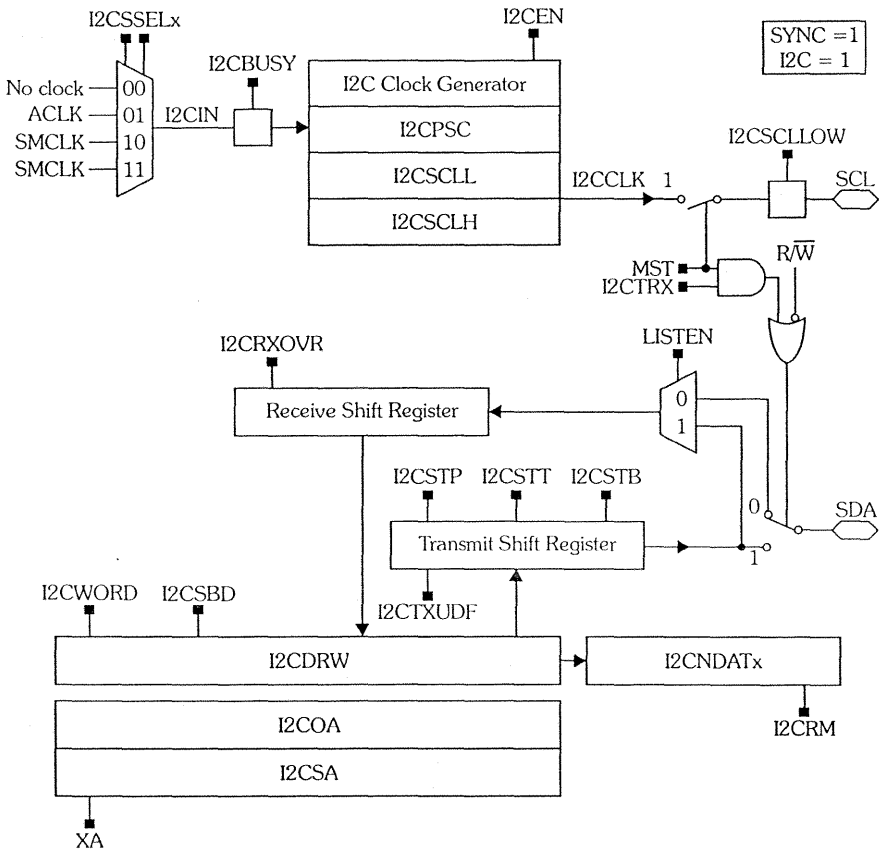


Figura 5-32

A operação da USART no modo I^2C é selecionada pelos *bits* UxCTL:SYNC = 1 e UxCTL:I2C = 1.

O módulo I^2C dos MSP430 pode funcionar em quatro modos de comunicação: transmissor mestre, receptor mestre, transmissor escravo e receptor escravo.

A seleção entre a operação no modo mestre ou escravo é feita pelo *bit* U0CTL:MST: que em nível "0" temos o modo escravo (a linha SCL é uma entrada e o *clock* é gerado pelo mestre do barramento), com MST em nível "1" temos o modo mestre (a linha SCL é uma saída e o *clock* é gerado pela USART).

A seleção entre transmissão e recepção é feita pelo *bit* I2CTCTL:TRX, mas depende também do modo de operação da I^2C: quando opera no modo mestre e com TRX = 0, temos o modo de recepção (a linha SDA é uma entrada), com TRX=1 temos o modo de transmissão (a linha SDA é uma saída).

Quando o módulo I^2C está configurado para o modo escravo, a seleção entre o modo de transmissão ou recepção é feita de acordo com o estado do *bit* R/W recebido juntamente com o endereço do dispositivo escravo. Quando R/W = 1, é selecionado o modo de recepção e quando R/W = 0, é selecionado o modo de transmissão.

5.10.3. Gerador de *Clock*

O circuito gerador de *clock* é utilizado para fornecer o sinal de *clock* quando o módulo I^2C opera no modo mestre.

São utilizados três registradores para o controle do sinal de *clock*:

- I2CPSC, para o controle do divisor de entrada (*prescaler*) que realiza a divisão do sinal de *clock* de entrada (ACLK ou SMCLK) por um fator de 1 a 256.
- I2CSCLH, que define o número de ciclos I2CPSC que compõem o ciclo alto do sinal de *clock* SCL. O número de ciclos é igual a I2CSCLH mais um.
- I2CSCLL, que define o número de ciclos I2CPSC que compõem o ciclo baixo do sinal de *clock* SCL. O número de ciclos é igual a I2CSCLL mais um.

A figura 5-33 representa os sinais de *clock* relacionados ao módulo I^2C.

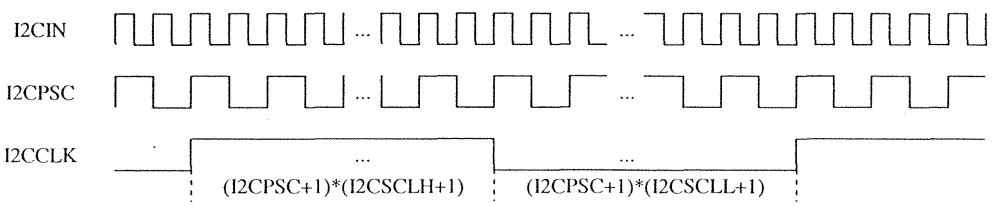


Figura 5-33

O fabricante recomenda que o registrador I2CPSC não seja carregado com um valor maior que 4. Caso esta recomendação não seja seguida, a USART pode apresentar um funcionamento incorreto.

5.10.4. Configuração da USART

Para configurar a USART para operar no modo I^2C, é importante seguir estes passos:

1. Configurar os pinos da USART para a função alternativa (veja a tabela 5-23).
2. Setar o *bit* de *reset* da USART (U0CTL:SWRST = 1).
3. Selecionar o modo I^2C (U0CTL:SYNC e U0CTL:I2C = 1). A partir desse momento, os registradores da USART passam a trabalhar no modo I^2C.
4. Apagar o *bit* de habilitação da I^2C (U0CTL:I2CEN = 0). Isso faz com que os bits I2CTRX (modo transmissor), I2CSTB (modo *byte* de partida), I2CSTP (emissão de *bit* de parada) e I2CSTT (emissão de *bit* de partida), todos localizados no registrador I2CTCTL, sejam apagados. O conteúdo dos registradores I2CDCTL e I2CDRW/I2CDRB também é apagado.
5. Configurar o modo de funcionamento do módulo I^2C (registradores U0CTL e I2CTCTL). A velocidade de comunicação no modo mestre é configurada pelos

registradores I2CPSC, I2CSCLH e I2CSCLL. Também é necessário configurar o endereço do dispositivo (registrador I2COA) e do dispositivo escravo com o qual deseja se comunicar (registrador I2CSA).

6. Após a configuração do módulo, habilita-o pelo *bit* U0CTL:I2CEN.

5.10.5. Operação no Modo Mestre

Para operação no modo mestre, é necessário setar o *bit* U0CTL:MST. Quando operando no modo mestre, o sinal de *clock* do barramento I^2C (SCL) é fornecido pela USART.

O módulo I^2C pode funcionar como transmissor ou receptor. Essa operação é controlada pelo *bit* I2CTCTL:I2CTRX. Quando está em nível "0", temos o modo de recepção e em nível "1", temos o modo de transmissão.

Quando funciona como transmissor, existem ainda duas possibilidades de operação: utilizar a contagem automática do número de *bytes* transmitidos (*bit* I2CTCTL:I2CRM = 0) ou o controle manual dessa operação (*bit* I2CTCTL:I2CRM = 1).

O controle automático do número de *bytes* transmitidos utiliza um registrador (I2CNDAT) para armazenar a quantidade total de *bytes* a ser transmitida (excluindo-se o primeiro de controle). Um contador interno é inicializado com o valor desse registrador e decrementado a cada *byte* transmitido. Quando o contador chega a zero, a transmissão pode ser automaticamente finalizada por uma condição de parada.

Os passos para realizar uma transmissão utilizando o modo mestre com contagem automática dos *bytes* transmitidos são os seguintes:

1. Aguarda-se que o módulo I^2C esteja liberado (*bit* I2CDCTL:I2CBUSY = 0).
2. Configura-se o módulo para o modo de transmissão (I2CTCTL:I2CTRX = 1).
3. A quantidade de *bytes* de dados a serem transmitidos deve ser carregada no registrador I2CNDAT.
4. Apaga-se o *flag* de interrupção de transmissão (I2CIFG:TXRDYIFG) e habilita-se a interrupção de transmissão do módulo (I2CIE:TXRDYIE = 1).
5. Ao setar o *bit* I2CTCTL:I2CSTT, a transmissão tem início. Primeiramente é transmitido o endereço do escravo (1 ou dois *bytes*, conforme a modalidade de endereçamento utilizada) mais o *bit* R/W. Repare que, emitida a condição de partida, o *bit* I2CSTT é automaticamente apagado pelo *hardware*.

Caso a fonte de clock da USART esteja inativa no instante em que o bit I2CTCTL:I2CSTT é setado, ela é temporariamente ativada, de forma a fornecer o sinal de clock para o módulo até que transmissão seja completada. Durante esse período, todos os periféricos ativos que utilizem essa fonte de clock também podem funcionar normalmente.

6. Após o início da transmissão, o *flag* I2CDCTL:I2CBB é setado indicando que o barramento I^2C encontra-se ocupado e o *flag* I2CDCTL:I2CBUSY é setado indicando que a USART encontra-se ocupada.
7. Após a transmissão de cada *byte*, o transmissor mantém a linha SDA livre no nono ciclo de *clock*, de forma que o receptor possa reconhecer a informação (mantendo-a em nível "0", gerando o sinal ACK). Caso o receptor reconheça a transmissão (ACK = 0), o transmissor dá prosseguimento a ela. Caso o receptor não reconheça a transmissão

por algum motivo (ACK = 1), o transmissor aborta a transmissão e seta o *flag* de não-reconhecimento (I2CIFG:NACKIFG).

8. Transmitido o endereço, a interrupção TXRDY é disparada e a RTI deve providenciar a carga do dado a ser transmitido no registrador I2CDRW (quando opera no modo 16 *bits*) ou I2CDRB (modo de 8 *bits*). O contador de dados transmitidos é decrementado automaticamente a cada transmissão.
9. Com o contador de *bytes* transmitidos no valor 0, a transmissão é automaticamente encerrada com a emissão de uma condição de parada (desde que o *bit* I2CTCTL:I2CSTP tenha sido setado antes de escrita do último valor no registrador de dados I2CDRW/I2CDRB).
10. Também é possível emitir uma condição de reinício (*restart*), em que a direção do pino SDA é invertida. Para isso basta setar novamente o *bit* I2CTCTL:I2CSTT.
11. Encerrada a transmissão (com uma condição de parada), os *flags* I2CSTP, I2CMST, I2CBB e I2CBUSY são automaticamente apagados.

A operação sem a utilização do contador de *bytes* segue basicamente o mesmo procedimento com as seguintes diferenças:

1. O *software* deve controlar a quantidade de *bytes* transmitidos (o que na prática permite mais do que 256 *bytes* em uma transmissão, que é o limite imposto pelo uso do registrador I2CNDAT).
2. A condição de parada não é gerada automaticamente. Sendo assim, o *software* deve setar o *bit* I2CTCTL:I2CSTP somente antes do início da transmissão do último *byte*.

Caso o registrador de dados da I^2C (I2CDRW ou I2CDRB) esteja vazio no instante em que é feita a carga do registrador de deslocamento do I^2C, ocorre um erro de buffer underflow e o flag I2CDCTL:I2CTXUDF é setado.

A recepção de um dado no modo mestre é realizada sempre após uma transmissão prévia. Para sinalizar ao receptor que o transmissor deseja receber dados, é necessária a transmissão do endereço do escravo com o *bit* R/W setado. A recepção no modo mestre também pode utilizar a contagem automática de *bytes* (registrador I2CNDAT). Os passos para realizar uma recepção no modo mestre são os seguintes:

1. Seleciona-se o modo de recepção (I2CTCTL:I2CTRX = 0).
2. Seta-se o *bit* de partida (I2CTCTL:I2CSTT), fazendo com que tenha início a transmissão do endereço do escravo com o *bit* R/W setado. O *bit* I2CSTT é automaticamente apagado pelo *hardware*. Os *flags* I2CDCTL:I2CBB e I2CDCTL:I2CBUSY são setados pelo *hardware*.
3. Após a transmissão do endereço, a linha SDA do mestre passa a funcionar como uma entrada e o dispositivo escravo passa a enviar o primeiro dado.
4. Caso seja utilizado o contador de dados (I2CNDAT), a cada caractere recebido, o contador é decrementado e o *flag* I2CIFG:RXRDYIFG é setado.
5. O dado recebido é armazenado no registrador I2CDRW/I2CDRB. Após a recepção de um caractere, o mestre deve gerar o sinal de reconhecimento (ACK=0), caso o caractere seja validado.
6. Caso o contador de *bytes* recebidos esteja sendo utilizado e o *bit* I2CSTP esteja setado, após a recepção do último caractere, uma condição de parada é gerada automati-

camente. Caso o *bit* I2CSTP esteja apagado ou não esteja sendo utilizado o contador de *bytes*, o *software* do usuário deve providenciar a geração da condição de parada.

7. Após a emissão da condição de parada, os *bits* I2CBB, I2CSTP, I2CMST e I2CBUSY são automaticamente apagados.

Caso um novo dado seja recebido, sem que o anterior tenha sido lido do registrador I2CDRW/I2CDRB, o flag I2CDCTL:I2CRXOVR é setado de forma a indicar a condição de buffer overflow.

5.10.5.1. Arbitragem

Uma condição especial ocorre quando, num barramento I^2C, dois ou mais dispositivos mestres tentam ocupar o barramento simultaneamente.

Neste caso, tem início um procedimento conhecido como arbitragem de barramento, em que os diversos dispositivos mestres concorrem pela "posse" do barramento.

O procedimento é relativamente simples: cada dispositivo mestre verifica a linha SDA após a escrita de um *bit*. Caso o barramento se encontre em nível diferente daquele em que foi escrito, o *flag* ALIFG é setado, indicando a perda de arbitramento de barramento. A transmissão em andamento é abortada e o módulo I^2C é comutado para o modo de recepção escrava.

Esse procedimento privilegia os dispositivos que estejam se comunicando com escravos com endereços baixos, ou que estejam transmitindo valores menores.

5.10.6. Operação no Modo Escravo

A operação no modo escravo é selecionada com o *bit* U0CTL:MST = 0. Nessa modalidade, o sinal de *clock* do barramento I^2C é fornecido pelo dispositivo mestre e a USART configura o pino SCL como uma entrada.

Normalmente, um dispositivo I^2C escravo inicia a sua operação configurado para o modo de recepção. A transição do modo de recepção para o modo de transmissão é feita automaticamente ao receber um *bit* R/W configurado em nível "1".

Os passos para efetuar a recepção em modo escravo são os seguintes:

1. O receptor aguarda a recepção de uma condição de partida.

Caso a fonte de clock da USART esteja inativa no instante em que a condição de partida é detectada, ela é temporariamente ativada, de forma a fornecer o sinal de clock para o módulo até a que recepção seja completada. Durante esse período, todos os periféricos ativos que utilizem essa fonte de clock também podem funcionar normalmente. Observe que o módulo I^2C mantém a linha de clock (SCL) em nível "0" até que a fonte de clock esteja ativa.

2. Após a condição de partida, os *flags* I2CDCTL:I2CBUSY, I2CDCTL:I2CBB e I2CIFG:STTIFG são setados.
3. Após a recepção do endereço enviado pelo dispositivo mestre, ele é comparado com o endereço do módulo (armazenado em I2COA). Caso os endereços não coincidam, a USART retorna ao estado de repouso, aguardando uma nova condição de partida.

4. Caso os endereços sejam idênticos, a USART envia um sinal de reconhecimento (ACK=0) e o *flag* I2CIFG:OAIFG é setado, indicando que a mensagem recebida é direcionada ao dispositivo escravo em questão. Repare que no caso de uma condição de reinício (*restart*) o *flag* OAIFG não é setado.
5. Tem início a recepção dos *bytes* de dados, iniciando pelo *byte* menos significativo (no caso do modo de 16 *bits*).
6. O dado recebido é armazenado no registrador I2CDRW/I2CDRB.
7. A cada *byte* recebido a USART envia um sinal de reconhecimento (ACK=0).
8. Ao ser detectada uma condição de parada, a recepção é encerrada e os *flags* I2CDCTL:I2CBB e I2CDCTL:I2CBUSY são automaticamente apagados.

Caso um novo dado seja recebido, sem que o anterior tenha sido lido do registrador I2CDRW/I2CDRB, o flag I2CDCTL:I2CRXOVR é setado de forma a indicar a condição de buffer overflow.

No caso de uma transmissão no modo escravo, ela somente tem início após a recepção do endereço escravo com o *bit* R/W apagado.

Neste caso, após o *flag* de coincidência de endereço (I2CIFG:OAIFG) ser setado, a USART carrega o dado proveniente do registrador de dados I^2C (I2CDRW ou I2CDRB) e em seguida tem início a transmissão dele (iniciando pelo *byte* menos significativo, no caso da transmissão de 16 *bits*).

Após a transmissão de cada *byte* a USART aguarda um sinal de reconhecimento. Caso o dado seja reconhecido (ACK=0), verifica-se a existência de uma condição de parada, caso não exista, o módulo aguarda a carga de um novo valor no registrador I2CDRW/I2CDRB de forma a realizar uma nova transmissão.

Caso o dado não seja reconhecido (ACK=1), o *flag* I2CIFG:NACKIFG é setado e a transmissão é encerrada.

Caso seja recebida uma condição de parada, os *flags* I2CDCTL:I2CBB e I2CDCTL:I2CBUSY são automaticamente apagados e a transmissão é encerrada.

Em ambos os casos, a USART retorna ao modo de recepção escrava.

Caso o registrador de dados da I^2C (I2CDRW ou I2CDRB) esteja vazio no instante em que é feita a carga do registrador de deslocamento do I^2C, ocorre um erro de buffer underflow e o flag I2CDCTL:I2CTXUDF é setado.

5.10.7. Operação com DMA

O módulo I^2C permite que se utilize o módulo DMA para realizar a transferência de dados recebidos/transmitidos entre a USART e a memória RAM do microcontrolador.

É possível utilizar o DMA tanto para automatizar a tarefa de transferência dos dados recebidos para a memória RAM (quando o *bit* U0CTL:RXDMAEN está setado) quanto da memória RAM para o registrador de dados no caso de uma transmissão (quando o *bit* U0CTL:TXDMAEN está setado).

Em ambos os casos, com o DMA ativado, o respectivo *flag* de interrupção (RXRDYIFG ou TXRDYIFG) torna-se inoperante, não sendo capaz de alterar o gerador de vetor de interrupção (I2CIV) e não é avaliado pelo controlador de interrupções.

5.10.8. Interrupções

O módulo I^2C dispõe de uma grande quantidade de interrupções. Todas elas compartilham o mesmo vetor (de número 8), conforme a tabela 5-1.

Existem, ao todo, oito diferentes eventos de interrupção relacionados ao módulo I^2C:

ALIFG - perda de arbitramento. Esse *flag* no dispositivo mestre perde uma disputa de arbitramento de barramento. Ele também é setado quando se tenta iniciar uma transferência I2C enquanto o *flag* I2CDCTL:I2CBB está setado. Quando esse *flag* é setado, os *bits* U0CTL:MST e I2CTCTL:I2CSTT são automaticamente apagados.

NACKIFG - não-reconhecimento. Setado quando o *bit* ACK = 1.

OAIFG - coincidência de endereço. Ocorre quando o endereço recebido no modo escravo coincide com o endereço do dispositivo (armazenado no registrador I2COA).

ARDYIFG - registradores prontos para acesso. Esse *flag*, quando setado, indica que os registradores da I^2C podem ser acessados. Essa situação ocorre sempre que uma transferência I^2C é completada.

RXRDYIFG - recepção de dados. Esse *flag* indica que um novo dado foi recebido pela interface I^2C. O *flag* é automaticamente apagado ao efetuar a leitura do registrador I2CDRW/I2CDRB.

TXRDYIFG - transmissão de dados. Esse *flag* é setado quando a interface I^2C está pronta para transmitir um novo dado (modo de transmissão mestre) ou quando um mestre está requisitando uma transmissão (modo de transmissão escrava). Em ambos os casos, o *flag* é apagado após a escrita do dado no registrador I2CDRW/I2CDRB.

GVIFG - chamada geral. Esse *flag* é setado quando um código de chamada geral (*general call*) é recebido pela I^2C.

STTIFG - condição de partida detectada. Esse *flag* é setado ao ser detectada uma condição de partida, quando o módulo I^2C está operando no modo escravo.

Os eventos listados podem ser individualmente habilitados no registrador I2CIE e o estado dos *flags* pode ser lido por meio do registrador I2CIFG. Adicionalmente, o módulo I^2C conta também com um gerador de vetor de interrupção (I2CIV), que pode ser utilizado para acelerar o processo de decodificação e tratamento das interrupções.

5.10.9. Conexões da USART

Os sinais I^2C estão disponíveis nos seguintes pinos:

Sinal	Pino
SDA	P3.1
SCL	P3.3

Tabela 5-23

5.10.10. Registradores da USART no Modo I^2C

Quando a USART opera no modo I^2C, utiliza os seguintes registradores para o seu funcionamento:

U0CTL - para configuração geral da USART.

I2CTCTL - controle de transferência da interface I^2C.

I2CDCTL - controle de dados I^2C.

I2CPSC, I2CSCLH e **I2CSCLL** - controle da velocidade de transmissão no modo mestre.

I2CDRW/I2CDRB - dado transmitido/recebido.

I2CNDAT - contagem de dados.

I2COA - endereço do dispositivo.

I2CSA - endereço do dispositivo escravo.

I2CIE - habilitação das interrupções da I^2C.

I2CIFG - *flags* de interrupção da I^2C.

I2CIV - gerador de vetor de interrupção da I^2C.

P3SEL - seleção da função alternativa para os pinos do módulo I^2C.

5.10.10.1. U0CTL

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0070	U0CTL	Leitura	RXDMA	TXDMA	I2C	XA	LISTEN	SYNC	MST	I2CEN
		Escrita	EN	EN						
		Reset	0	0	0	0	0	0	0	1

RXDMAEN - habilitação do DMA para recepção I^2C. Com esse *bit* habilitado, o controlador de DMA pode transferir automaticamente os dados recebidos para a memória do microcontrolador. Neste caso, o *flag* I2CIE:RXRDYIE é ignorado.

0 - DMA de recepção desabilitado;

1 - DMA de recepção habilitado (símbolo **RXDMAEN**).

RXDMAEN - habilitação do DMA para transmissão I^2C. Com esse *bit* habilitado, o controlador de DMA pode transferir automaticamente dados da memória do microcontrolador para o registrador de dados do I^2C. Neste caso, o *flag* I2CIE:TXRDYIE é ignorado.

0 - DMA de recepção desabilitado;

1 - DMA de recepção habilitado (símbolo **TXDMAEN**).

I2C - ativação do modo I^2C (somente na USART0 dos modelos 15x e 16x):

0 - modo SPI;

1 - modo I^2C (símbolo **I2C**).

XA - endereçamento estendido:

0 - 7 *bits*;

1 - 10 *bits* (símbolo **XA**).

LISTEN - habilitação do *loopback* (a transmissão e a recepção são internamente conectadas):

0 - modo normal;

1 - modo *loopback*, os dados transmitidos são internamente direcionados para a etapa de recepção (válido somente quando operando no modo transmissor mestre (U0CTL:MST e I2CTCTL:I2CTRX = 1)) (símbolo **LISTEN**).

SYNC - seleção do modo síncrono:

0 - modo assíncrono;

1 - modo síncrono (SPI ou I^2C) (símbolo **SYNC**).

MST - seleção do modo mestre (apagado automaticamente no caso de uma perda de arbitramento de barramento ou quando uma condição de parada (*STOP*) é gerada):

0 - modo escravo;

1 - modo mestre (símbolo **MST**).

I2CEN - habilitação do módulo I^2C. Quando o *bit* I2C está apagado, esse *bit* encontra-se setado e operando na função SWRST. Quando o *bit* I2C é setado pela primeira vez após um *reset*, esse *bit* passa a operar na função I2CEN e é apagado.

0 - módulo I^2C desabilitado;

1 - módulo I^2C habilitado (símbolo **I2CEN**).

5.10.10.2. I2CTCTL ou U0TCTL

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0071	I2CTCTL U0TCTL	Leitura Escrita	I2CWORD*	I2CRM*	I2CSSEL*		I2CTRX	I2CSTB	I2CSTP	I2CSTT
		Reset	0	0	0	0	0	0	0	0

*** Estes *bits* somente podem ser modificados quando U0CTL:I2CEN = 0.**

I2CWORD - configuração da I^2C para operação com *byte* ou *word*:

0 - *byte*;

1 - *word* (símbolo **I2CWORD**).

I2CRM - modo de repetição:

0 - o número de *bytes* de dados a serem transmitidos é especificado pelo registrador I2CNDAT;

1 - o número de *bytes* transmitidos é especificado pelo *software* e o registrador I2CNDAT não é utilizado (símbolo **I2CRM**).

I2CSSELx - seleção da fonte de *clock* para o módulo I^2C (símbolos **SSEL1** e **SSEL0**):

00 - *clock* desligado (módulo I^2C inativo);

01 - ACLK;

10 - SMCLK;

11 – SMCLK.

I2CTRX - seleção do modo de transmissão ou recepção quando operando no modo mestre (a seleção entre transmissão/recepção no modo escravo é selecionada pelo *bit* R/W no *byte* de endereçamento recebido). Quando operando no modo escravo, esse *bit* deve ser mantido em "0":

0 - modo de recepção;

1 - modo de transmissão (símbolo **I2CTRX**).

I2CSTB - envia um *byte* de partida (endereço = 0 e R/W=1) (quando operando no modo mestre (U0CTL:MST=1) e desde que o *bit* de partida I2CSTT esteja setado). Uma vez que a transmissão do *byte* de partida tenha início, esse *bit* é automaticamente apagado.

0 - nenhuma operação;

1 - envia um *bit* de partida seguido de um *byte* de partida, sem *STOP* (símbolo **I2CSTB**).

I2CSTP - envia uma condição de parada. Após ela, esse *bit* é automaticamente apagado.

0 - nenhuma ação;

1 - envia uma condição de parada (*STOP*) (símbolo **I2CSTP**).

I2CSTT - envia uma condição de partida. Após o seu envio, esse *bit* é automaticamente apagado.

0 - nenhuma ação;

1 - envia uma condição de partida (*START*) (símbolo **I2CSTT**).

5.10.10.3. I2CDCTL ou U0RCTL

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0072	I2CDCTL U0RCTL	Leitura	Não utilizado	Não utilizado	I2CBUSY	I2C SCLLOW	I2CSBD	I2C TXUDF	I2C RXOVR	I2CBB
		Reset	0	0	0	0	0	0	0	0

I2CBUSY - indica que a USART está ocupada em uma operação de transmissão ou recepção:

0 - USART livre;

1 - USART ocupada (símbolo **I2CBUSY**).

I2CSCLLOW - indica se um dispositivo escravo está mantendo a linha de *clock* (SCL) em nível "0":

0 - linha de *clock* normal;

1 - linha de *clock* sendo mantida em nível "0" (símbolo **I2CSCLLOW**).

I2CSBD - recepção de *byte* simples. Indica que o registrador I2CDRW (quando operando no modo 16 *bits*) contém uma *word* completa ou somente um *byte*:

0 - o registrador I2CDRW contém uma *word* completa;

1 - o registrador I2CDRW contém apenas o *byte* menos significativo(símbolo **I2CSBD**).

I2CTXUDF - indicador de erro de esvaziamento do *buffer* de transmissão:

0 - não ocorreu erro de esvaziamento;

1 - ocorreu um erro de esvaziamento do buffer de transmissão (símbolo **I2CTXUDF**).

I2CRXOVR - indicador de sobrescrita do dado no *buffer* de recepção:

0 - não ocorreu erro;

1 - um novo caractere foi escrito no I2CDRW/I2CDRB antes da leitura do anterior (símbolo **I2CRXOVR**).

I2CBB - indicador de barramento I^2C ocupado. Esse *bit* é setado por uma condição de partida e apagado por uma condição de parada, ou quando U0CTL:I2CEN = 0:

0 - barramento I^2C livre;

1 - barramento I^2C ocupado (símbolo **I2CBB**).

5.10.10.4. I2CPSC ou U0MCTL

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0073	I2CPSC U0MCTL	Leitura / Escrita	Fator de divisão da entrada de *clock* (I2CIN) do módulo I2C. O fator de divisão é igual a I2CPSC + 1. Fatores maiores que 4 não são recomendados.							
		Reset	0	0	0	0	0	0	0	0

*** Esse registrador somente pode ser modificado quando I2CEN =0.**

5.10.10.5. I2CSCLH ou U0BR0

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0074	I2CSCLH U0BR0	Leitura / Escrita	Determina o período alto do sinal de *clock* SCL. O período alto de SCL é igual a (I2CSCLH+2)*(I2CPSC+1)							
		Reset	?	?	?	?	?	?	?	?

*** Esse registrador somente pode ser modificado quando I2CEN =0.**

0 - período alto do SCL = 5 * (I2CPSC+1)
1 - período alto do SCL = 5 * (I2CPSC+1)
2 - período alto do SCL = 5 * (I2CPSC+1)
3 - período alto do SCL = 5 * (I2CPSC+1)
4 - período alto do SCL = 6 * (I2CPSC+1)
...
255 - período alto do SCL = 257 * (I2CPSC+1)

5.10.10.6. I2CSCLL ou U0BR1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0075	I2CSCLL U0BR1	Leitura / Escrita	Determina o período baixo do sinal de *clock* SCL. O período baixo de SCL é igual a (I2CSCLL+2)*(I2CPSC+1)							
		Reset	?	?	?	?	?	?	?	?

*** Esse registrador somente pode ser modificado quando I2CEN =0.**

0 - período baixo do SCL = 5 * (I2CPSC+1)
1 - período baixo do SCL = 5 * (I2CPSC+1)
2 - período baixo do SCL = 5 * (I2CPSC+1)
3 - período baixo do SCL = 5 * (I2CPSC+1)
4 - período baixo do SCL = 6 * (I2CPSC+1)
...
255 - período baixo do SCL = 257 * (I2CPSC+1)

5.10.10.7. I2CDRW

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0076	I2CDRW	Leitura	I2CDRW x							
		Escrita								
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	I2CDRW x							
		Escrita								
		Reset	0	0	0	0	0	0	0	0

O registrador I2CDRW é utilizado para transmissão e recepção de dados pela interface I^2C, quando está configurada para operar no modo de 16 *bits* (I2CTCTL:I2CWORD = 1). Neste caso, esse registrador somente deve ser acessado por instruções do tipo *word*.

5.10.10.8. I2CDRB

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0076	I2CDRB	Leitura	I2CDRBx							
		Escrita								
		Reset	?	?	?	?	?	?	?	?

O registrador I2CDRB é utilizado para transmissão e recepção de dados pela interface I^2C, quando está configurada para operar no modo de 8 *bits* (I2CTCTL:I2CWORD = 0). Neste caso, esse registrador somente deve ser acessado por instruções do tipo *byte*.

5.10.10.9. I2COA

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0118	I2COA	Leitura	I2COA x							
		Escrita								
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	I2COA x							
		Escrita								
		Reset	0	0	0	0	0	0	0	0

*** Este registrador somente pode ser modificado quando I2CEN =0.**

O registrador I2COA contém o endereço local do dispositivo I^2C. Quando operando no modo de 7 *bits*, os *bits* 7 até 15 não podem ser modificados e permanecem sempre com o valor "0". No modo de 10 *bits*, os *bits* 10 até 15 não podem ser modificados.

5.10.10.10. I2CSA

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x011A	I2CSA	Leitura / Escrita	I2CSA x							
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura / Escrita	I2CSA x							
		Reset	0	0	0	0	0	0	0	0

O registrador I2CSA contém o endereço do dispositivo I^2C escravo para o qual serão destinadas as informações. Quando operando no modo de 7 *bits*, os *bits* 7 até 15 não podem ser modificados e permanecem sempre com o valor "0". No modo de 10 *bits*, os *bits* 10 até 15 não podem ser modificados.

5.10.10.11. I2CNDAT

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0052	I2CNDAT	Leitura / Escrita	I2CNDATx							
		Reset	0	0	0	0	0	0	0	0

O registrador I2CNDAT contém o número de *bytes* de dados a serem transmitidos ou recebidos.

5.10.10.12. I2CIE

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0050	I2CIE	Leitura / Escrita	STTIE	GCIE	TXRDYIE	RXRDYIE	ARDYIE	OAIE	NACKIE	ALIE
		Reset	0	0	0	0	0	0	0	0

STTIE - habilitação da interrupção por recepção de condição de partida (símbolo **STTIE**).
GCIE - habilitação da interrupção por recepção de chamada geral (símbolo **GCIE**).
TXRDYIE - habilitação da interrupção de *buffer* de transmissão vazio (símbolo **TXRDYIE**).
RXRDYIE - habilitação da interrupção por recepção de dado (símbolo **RXRDYIE**).
ARDYIE - habilitação da interrupção de registradores prontos para acesso (símbolo **ARDYIE**).
OAIE - habilitação da interrupção de coincidência de endereço (símbolo **OAIE**).
NACKIE - habilitação da interrupção por recepção de NACK (símbolo **NACKIE**).
ALIE - habilitação da interrupção por perda de arbitramento (símbolo **ALIE**).

5.10.10.13. I2CIFG

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0051	I2CIFG	Leitura / Escrita	STTIFG	GCIFG	TXRDY IFG	RXRDY IFG	ARDYIFG	OAIFG	NACKIFG	ALIFG
		Reset	0	0	0	0	0	0	0	0

STTIFG - sinalizador da interrupção por recepção de condição de partida (símbolo **STTIFG**).
GCIFG - sinalizador da interrupção por recepção de chamada geral (símbolo **GCIFG**).
TXRDYIFG - sinalizador da interrupção de *buffer* de transmissão vazio (símbolo **TXRDYIFG**).
RXRDYIFG - sinalizador da interrupção por recepção de dado (símbolo **RXRDYIFG**).
ARDYIFG - sinalizador da interrupção de registradores prontos para acesso (símbolo **ARDYIFG**).
OAIFG - sinalizador da interrupção de coincidência de endereço (símbolo **OAIFG**).
NACKIFG - sinalizador da interrupção por recepção de NACK (símbolo **NACKIFG**).
ALIFG - sinalizador da interrupção por perda de arbitramento (símbolo **ALIFG**).

5.10.10.14. I2CIV

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x011C	I2CIV	Leitura	0	0	0	0	0	0	0	0
		Escrita	-	-	-	-	-	-	-	-
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	0	0	0	I2CIVx				0
		Escrita	-	-	-					-
		Reset	0	0	0	0	0	0	0	0

I2CIV	Fonte de interrupção	*Flag* de interrupção	Prioridade
0x00	Nenhuma interrupção	-	
0x02	Perda de arbitragem do barramento	I2CIFG:ALIFG	Maior
0x04	Não-reconhecimento	I2CIFG:NACKIFG	
0x06	Próprio endereço	I2CIFG:OAIFG	
0x08	Registradores prontos para acesso	I2CIFG:ARDYIFG	
0x0A	Recepção de dado	I2CIFG:RXRDYIFG	
0x0C	*Buffer* de transmissão vazio	I2CIFG:TXRDYIFG	
0x0E	Chamada geral	I2CIFG:GCIFG	
0x10	Recepção de condição de partida	I2CIFG:STTIFG	Menor

Tabela 5-24

5.10.11. Exemplos de Utilização

O exemplo seguinte demonstra a utilização do módulo I^2C de um *chip* MSP430F169 comunicando-se com uma memória EEPROM I^2C do tipo 24C256. O circuito básico pode ser visto na figura 5-34. Os sinais I2C_SDA e I2C_SCL devem ser conectados aos pinos P3.1 e P3.3 respectivamente.

O programa foi baseado na nota de aplicação SLAA208 de Christian Hemitscheck *(Texas Instruments)* com a inclusão de funções para escrita e leitura de valores inteiros de 16 e 32 *bits*, além de valores ***float*** de 32 *bits*.

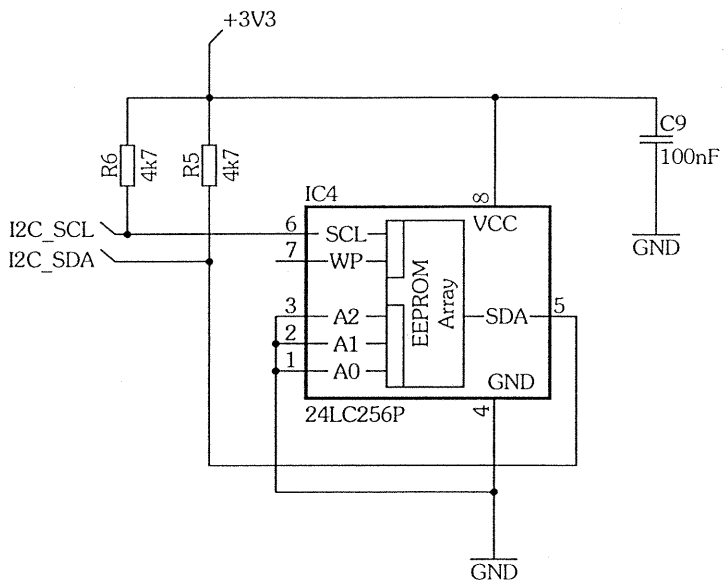


Figura 5-34

```
#include "io430x16x.h"
#include "intrinsics.h"

#define endereco_escravo   0x50;

int PtrTransmit;
unsigned char I2CBuffer[3];

//*********************************************************************
// Inicializa a USART para o modo I2C
//*********************************************************************
void usart_I2C_init(void)
{
   P3SEL_bit.P3SEL_1 = 1;    // pino P3.1 para função SDA
   P3SEL_bit.P3SEL_3 = 1;    // pino P3.3 para função SCL
   // inicializa a USART para o modo I2C
   U0CTL |= I2C+SYNC;
   U0CTL_bit.I2CEN = 0;      // desabilita o módulo I2C
   // seleciona o tamanho de palavra = byte, clock I2C = SMCLK
   I2CTCTL = I2CTRX+I2CSSEL_2;
   I2CSA = endereco_escravo; // endereço do dispositivo escravo
   I2COA = 0x01A5;  // endereço do dispositivo mestre
   I2CPSC = 0x00;            // Clock do I2C = SMCLK / 1
   I2CSCLH = 0x03;  // Período alto do SCL = 5*I2C clock
   I2CSCLL = 0x03;  // Período alto do SCL = 5*I2C clock
   U0CTL_bit.I2CEN = 1;      // habilita o módulo I2C
}

//*********************************************************************
// Escreve um dado no endereço especificado da EEPROM
//*********************************************************************

void EEPROM_bytewrite(unsigned int endereco, unsigned char dado)
{
   unsigned char end_hi, end_lo;
   // aguarda a interface I2C estar pronta
```

```
    while (U0RCTL_bit.I2CBUSY);
    end_hi = endereco >> 8;   // parte alta do endereço
    end_lo = endereco & 0xFF; // parte baixa do endereço
    // armazena o endereço e o dado no buffer I2C
    I2CBuffer[2] = end_hi;
    I2CBuffer[1] = end_lo;
    I2CBuffer[0] = dado;
    // seta o ponteiro de transmissão para apontar para o elemento 2 do buffer
    // de transmissão (end_hi)
    PtrTransmit = 2;
    U0CTL_bit.MST = 1;        // modo mestre
    U0TCTL_bit.I2CTRX = 1;    // modo de transmissão
    I2CIFG_bit.TXRDYIFG = 0;// apaga flag de transmissão
    I2CIE_bit.TXRDYIE = 1;    // habilita interrupção de transmissão
    // número de bytes a serem transmitidos =
    // 1 de controle + 2 de endereço e 1 de dado
    I2CNDAT = 3;
    // inicia a comunicação setando a condição de partida
    // a condição de parada também é setada e inserida ao término
    // da transmissão
    I2CTCTL |= I2CSTT+I2CSTP;
}

//*********************************************************************
// Lê um dado no endereço atual da EEPROM
//*********************************************************************
unsigned char EEPROM_byteread(void)
{
    // aguarda a interface I2C estar pronta
    while (U0RCTL_bit.I2CBUSY);
    U0TCTL_bit.I2CTRX = 0;    // modo de recepção
    I2CIE_bit.RXRDYIE = 1;    // habilita interrupção de recepção
    U0CTL_bit.MST = 1;        // modo mestre
    I2CNDAT = 1;              // recepção de 1 byte
    I2CIFG_bit.ARDYIFG = 0;   // apaga flag de acesso
    // inicia a recepção ativando a condição de partida
    // uma condição de reinicio e de parada são geradas após
    // a recepção
    I2CTCTL |= I2CSTT+I2CSTP;
    while (!I2CIFG_bit.ARDYIFG);//aguarda a recepção ser completada
    // retorna com o dado recebido
    return I2CBuffer[0];
}

//*********************************************************************
// Lê um dado no endereço especificado da EEPROM
//*********************************************************************

unsigned char EEPROM_byterandomread(unsigned int endereco)
{
    unsigned char end_hi, end_lo;
    // aguarda a interface I2C estar pronta
    while (U0RCTL_bit.I2CBUSY);
    end_hi = endereco >> 8;   // parte alta do endereço
    end_lo = endereco & 0xFF; // parte baixa do endereço
    I2CBuffer[1] = end_hi;
    I2CBuffer[0] = end_lo;
    // seta o ponteiro de transmissão para apontar para o elemento 1 do buffer
    // de transmissão (end_hi)
    PtrTransmit = 1;
    U0CTL_bit.MST = 1;        // modo mestre
    U0TCTL_bit.I2CTRX = 1;    // modo de transmissão
    I2CIFG_bit.TXRDYIFG = 0;  // apaga flag de transmissão
    I2CIE_bit.TXRDYIE = 1;    // habilita interrupção de transmissão
    // configura o número de bytes a serem transmitidos:
    // 1 de controle + 2 de endereço
    I2CNDAT = 2;
    I2CIFG_bit.ARDYIFG = 0;   // apaga flag de acesso
    U0TCTL_bit.I2CSTT = 1;    // condição de partida, inicia a comunicação
    while (I2CIFG_bit.ARDYIFG);//aguarda a transmissão ser completada
    U0TCTL_bit.I2CTRX = 0;    // modo de recepção
    I2CIE_bit.RXRDYIE = 1;    // habilita interrupção de recepção
    I2CNDAT = 1;              // recepção de 1 byte
```

```
    I2CIFG_bit.ARDYIFG = 0;   // apaga flag de acesso
    // inicia a recepção ativando a condição de partida
    // uma condição de reinicio e de parada são geradas após
    // a recepção
    I2CTCTL |= I2CSTT+I2CSTP;
    while (!I2CIFG_bit.ARDYIFG);//aguarda a recepção ser completada
    // retorna com o dado recebido
    return I2CBuffer[0];
}

//*********************************************************************
// Aguarda um ACK da EEPROM
//*********************************************************************
void EEPROM_ackpolling(void)
{
    // aguarda a interface I2C estar pronta
    while (U0RCTL_bit.I2CBUSY);
    // modifica a configuração do módulo I2C
    U0CTL_bit.I2CEN = 0;                // desabilita o módulo
    U0TCTL_bit.I2CRM = 1;               // transmissão controlada por software
    U0CTL_bit.I2CEN = 1;                // habilita módulo I2C
    I2CIFG = NACKIFG;                   // seta o flag de NACK
    while (I2CIFG_bit.NACKIFG)          // enquanto o NACK estiver setado
    {
            I2CIFG= 0;                  // apaga os flags de interrupção
            U0CTL_bit.MST = 1;          // modo mestre
            U0TCTL_bit.I2CTRX = 1;      // modo de transmissão
            U0TCTL_bit.I2CSTT = 1;      // envia START
            while (U0TCTL_bit.I2CSTT); // aguarda o envio do START
            U0TCTL_bit.I2CSTP = 1;      // envia STOP
            // aguarda a interface I2C estar pronta
            while (U0RCTL_bit.I2CBUSY);
    }
    U0CTL_bit.I2CEN = 0;        // desabilita o módulo
    U0TCTL_bit.I2CRM = 0;       // transmissão controlada pelo hardware
    U0CTL_bit.I2CEN = 1;        // habilita módulo I2C
}
//*********************************************************************
// RTI da I2C
//*********************************************************************
#pragma vector=USART0TX_VECTOR
__interrupt void ISR_I2C(void)
{
    switch (I2CIV)
    {
            case I2CIV_AL:    // perda de arbitragem
                              break;
            case I2CIV_NACK:// NACK
                              break;
            case I2CIV_OA:    // próprio endereço (own address)
                              break;
            case I2CIV_ARDY:// pronto para acesso
                              break;
            case I2CIV_RXRDY:// recepção
                              // armazena o dado no buffer
                              I2CBuffer[0] = I2CDRB;
                              break;
            case I2CIV_TXRDY:// transmissão
                              // armazena o dado do buffer (apontado pelo
                              // PtrTransmit) no registrador de dados da I2C
                              I2CDRB = I2CBuffer[PtrTransmit];
                              // se o ponteiro for maior que 0 decrementa o mesmo,
                              // caso seja igual a zero, desabilita a interrupção
                              // de transmissão I2C
                              if (PtrTransmit) PtrTransmit --;
                                    else I2CIE_bit.TXRDYIE = 0;
                              break;
            case I2CIV_GC:    // chamada geral (general call)
                              break;
            case I2CIV_STT:// condição de partida
                              break;
    }
}
```

```
//*********************************************************************
// Escreve um valor inteiro de 16 bits no endereço especificado
// da EEPROM
//*********************************************************************
void EEPROM_INTwrite(unsigned int endereco, unsigned int valor)
{
   unsigned char temp;
   temp = valor;     // temp = parte baixa de valor
   EEPROM_bytewrite(endereco,temp);
   EEPROM_ackpolling();      // aguarda o término da escrita
   // temp = parte alta de valor
   temp = valor>>8;
   EEPROM_bytewrite(endereco+1,temp);
   EEPROM_ackpolling();      // aguarda o término da escrita
}

//*********************************************************************
// Lê um valor inteiro de 16 bits do endereço especificado
// da EEPROM
//*********************************************************************
unsigned int EEPROM_INTrandomread(unsigned int endereco)
{
   unsigned int result;
   // lê o byte do endereço
   result = EEPROM_byterandomread(endereco);
   // lê o byte do próximo endereço, rotaciona 8 bits à esquerda
   // e soma ao resultado
   result+= EEPROM_byteread()<<8;
   // retorna o resultado
   return(result);
}

//*********************************************************************
// Escreve um valor inteiro de 32 bits no endereço especificado
// da EEPROM
//*********************************************************************
void EEPROM_LONGwrite(unsigned int endereco, unsigned long valor)
{
   // a união uvar32 aloca no mesmo espaço de memória, uma
   // variável inteira de 32 bits e 4 variáveis inteiras
   // de 8 bits
   struct svar32
   {
           char byte0;
           char byte1;
           char byte2;
           char byte3;
   };
   union uvar32
   {
           unsigned long int var32;
           struct svar32 var8;
   } temp;
   // atribui o valor à parte de 32 bits de temp
   temp.var32 = valor;
   // escreve o byte menos significativo
   EEPROM_bytewrite(endereco,temp.var8.byte0);
   EEPROM_ackpolling();      // aguarda o término da escrita
   // escreve o próximo byte
   EEPROM_bytewrite(++endereco,temp.var8.byte1);
   EEPROM_ackpolling();      // aguarda o término da escrita
   // escreve o próximo byte
   EEPROM_bytewrite(++endereco,temp.var8.byte2);
   EEPROM_ackpolling();      // aguarda o término da escrita
   // escreve o byte mais significativo
   EEPROM_bytewrite(++endereco,temp.var8.byte3);
   EEPROM_ackpolling();      // aguarda o término da escrita
}

//*********************************************************************
// Lê um valor inteiro de 32 bits no endereço especificado
// da EEPROM
//*********************************************************************
```

```
unsigned long EEPROM_LONGrandomread(unsigned int endereco)
{
   // a união uvar32 aloca no mesmo espaço de memória, uma
   // variável inteira de 32 bits e 4 variáveis inteiras
   // de 8 bits
   struct svar32
   {
            char byte0;
            char byte1;
            char byte2;
            char byte3;
   };
   union uvar32
   {
            unsigned long int var32;
            struct svar32 var8;
   } temp;
   // lê o byte do endereço e guarda no byte0
   temp.var8.byte0 = EEPROM_byterandomread(endereco);
   // lê o próximo byte e guarda em byte1
   temp.var8.byte1 = EEPROM_byteread();
   // lê o próximo byte e guarda em byte2
   temp.var8.byte2 = EEPROM_byteread();
   // lê o próximo byte e guarda em byte3
   temp.var8.byte3 = EEPROM_byteread();
   // retorna o valor de 32 bits
   return (temp.var32);
}

//*********************************************************************
// Escreve um valor float no endereço especificado da EEPROM
//*********************************************************************
void EEPROM_FLOATwrite(unsigned int endereco, float valor)
{
   // a união uvarfloat aloca no mesmo espaço de memória, uma
   // variável float e 4 variáveis inteiras de 8 bits
   struct svar32
   {
            char byte0;
            char byte1;
            char byte2;
            char byte3;
   };
   union uvarfloat
   {
            float varfloat;
            struct svar32 var8;
   } temp;
   temp.varfloat = valor;
   EEPROM_bytewrite(endereco,temp.var8.byte0);
   EEPROM_ackpolling();
   EEPROM_bytewrite(++endereco,temp.var8.byte1);
   EEPROM_ackpolling();
   EEPROM_bytewrite(++endereco,temp.var8.byte2);
   EEPROM_ackpolling();
   EEPROM_bytewrite(++endereco,temp.var8.byte3);
   EEPROM_ackpolling();
}



//*********************************************************************
// Lê um valor float do endereço especificado da EEPROM
//*********************************************************************
float EEPROM_FLOATrandomread(unsigned int endereco)
{
   // a união uvarfloat aloca no mesmo espaço de memória, uma
   // variável float e 4 variáveis inteiras de 8 bits
   struct svar32
   {
            char byte0;
            char byte1;
            char byte2;
```

```
        char byte3;
    };
    union uvarfloat
    {
        float varfloat;
        struct svar32 var8;
    } temp;
    temp.var8.byte0 = EEPROM_byterandomread(endereco);
    temp.var8.byte1 = EEPROM_byteread();
    temp.var8.byte2 = EEPROM_byteread();
    temp.var8.byte3 = EEPROM_byteread();
    return (temp.varfloat);
}
void main(void)
{
    volatile unsigned char Data[6];
    volatile unsigned long teste32;
    volatile float tempflt;
    WDTCTL = WDTPW+WDTHOLD;   // desliga o Watchdog
    usart_I2C_init();         // Inicializa o módulo I2C
    __enable_interrupt();
    EEPROM_bytewrite(0x0000,0xFA);
    EEPROM_ackpolling();      // Aguarda escrita na EEPROM
    EEPROM_bytewrite(0x0001,0xB1);
    EEPROM_ackpolling();      // Aguarda escrita na EEPROM
    EEPROM_bytewrite(0x0002,0x00);
    EEPROM_ackpolling();      // Aguarda escrita na EEPROM
    EEPROM_bytewrite(0x0003,0x12);
    EEPROM_ackpolling();      // Aguarda escrita na EEPROM
    EEPROM_bytewrite(0x0004,0x34);
    EEPROM_ackpolling();      // Aguarda escrita na EEPROM
    EEPROM_bytewrite(0x0005,0x56);
    EEPROM_ackpolling();      // Aguarda escrita na EEPROM
    // escreve o valor 0xABCD no endereço 0x0010
    EEPROM_INTwrite (0x0010,0xABCD);
    // escreve o valor 0x01234567 no endereço 0x0020
    EEPROM_LONGwrite (0x0020,0x01234567);
    tempflt = 12.3456;
    // escreve o valor 12.3456 no endereço 0x0030
    EEPROM_FLOATwrite(0x0030,tempflt);
    tempflt = 0;
    // Lê um byte no endereço 0x0000
    Data[0] = EEPROM_byterandomread(0x0000);
    // lê um byte no endereço 0x0001
    Data[1] = EEPROM_byteread();
    // lê um byte no endereço 0x0002
    Data[2] = EEPROM_byteread();
    // lê um byte no endereço 0x0003
    Data[3] = EEPROM_byteread();
    // lê um byte no endereço 0x0004
    Data[4] = EEPROM_byteread();
    // lê um byte no endereço 0x0005
    Data[5] = EEPROM_byteread();
    // lê um inteiro no endereço 0x0010
    teste32 = EEPROM_INTrandomread(0x0010);
    teste32++;
    // lê um inteiro longo no endereço 0x0020
    teste32 = EEPROM_LONGrandomread(0x0020);
    teste32++;
    // lê um float no endereço 0x0030
    tempflt = EEPROM_FLOATrandomread(0x0030);
    tempflt += 1;
    while (1);
}
```

Exemplo 5-12

5.11. Comparador Analógico

O comparador analógico (Comp_A) é um dispositivo utilizado para comparar duas tensões analógicas. A sua saída é colocada em nível lógico "1" quando a tensão na entrada positiva é maior que a tensão na entrada negativa. A saída vai a nível lógico "0" quando a tensão na entrada negativa é maior que a tensão na entrada positiva.

Esse tipo de circuito pode ser utilizado em diversas aplicações, tais como: conversões A/D simples, monitoração de tensões analógicas sem o uso de um ADC, etc.

O comparador analógico pode ser encontrado na maioria dos modelos MSP430, tais como: MSP430F11x1, F12x, F13x, F14x, F15x, F16x, F41x, FW42x, F43x, FW43x e F44x. Os *chips* dotados de conversores ADC de 10 e de 16 *bits* não possuem comparador analógico.

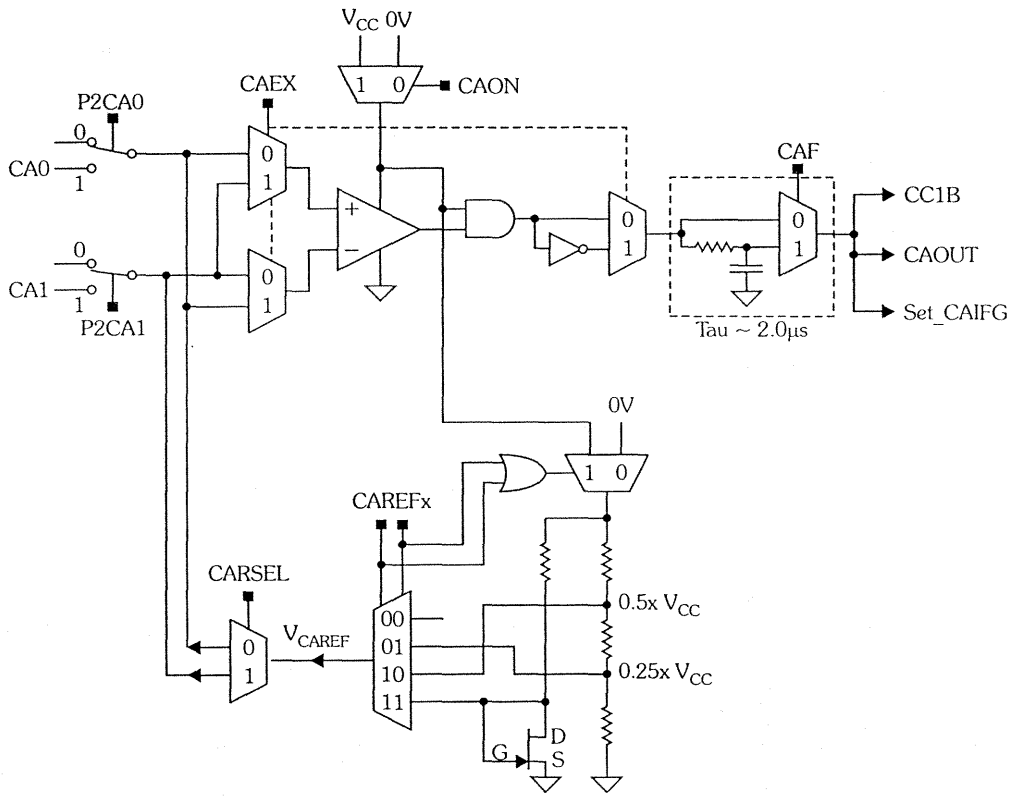


Figura 5-35

Algumas características do comparador analógico dos MSP430:

- Possibilidade de inversão por *software* dos terminais de entrada do comparador;
- Saída disponível externamente;
- Filtragem do sinal de saída do comparador selecionável por *software*;
- Capacidade de iniciar uma captura no *timer* A;
- Possibilidade de desativação dos *buffers* digitais nos pinos de entrada do comparador;

- Referência interna de tensão;
- Baixo consumo (tipicamente de 45μA para uma alimentação de 3,3V).

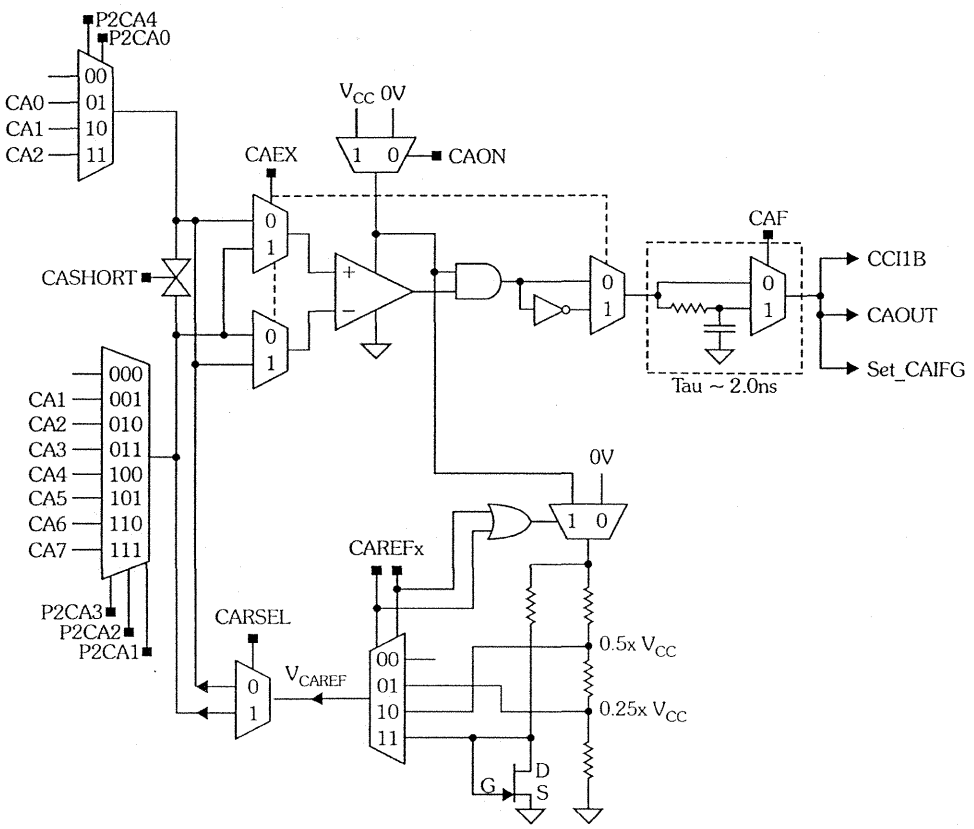


Figura 5-36

Os microcontroladores da família 2xx implementam uma nova versão do módulo comparador analógico (figura 5-36), denominada "comparador +", que inclui duas facilidades adicionais:

- Multiplexador analógico nas entradas, que permite selecionar um entre diversos pinos para cada entrada do comparador.
- Possibilidade de curto-circuitar internamente as entradas do comparador.

O funcionamento do comparador analógico é bastante simples: o "coração" do módulo é o circuito comparador de tensão analógico. As entradas desse circuito, doravante chamadas de E+ e E-, podem ser conectadas aos pinos externos do microcontrolador ou à referência interna de tensão, conforme veremos adiante.

Os pinos externos, chamados de CA0 e CA1, podem ser conectados ao comparador, desde que os *bits* CACTL2:P2CA0 e CACTL2:P2CA1 estejam setados. Quando um desses *bits* está apagado, o respectivo pino encontra-se desconectado da entrada do comparador.

Adicionalmente, o módulo comparador permite que se desativem os *buffers* digitais dos pinos da porta utilizada para o comparador. Para isso, basta setar o respectivo *bit* do registrador CAPD (*bit* 0 para o pino 0, *bit* 1 para o pino 1 e assim por diante). Essa facilidade permite que se

minimizem as correntes parasitas que fluem da entrada analógica para os *buffers* digitais do pino do *chip*, reduzindo o consumo de corrente do sistema.

Ainda na entrada do comparador, encontramos um multiplexador analógico, cuja função é permitir a inversão das entradas do comparador. O multiplexador é controlado pelo *bit* CACTL1:CAEX: quando CAEX=0, o pino CA0 pode estar conectado à entrada E+ e CA1 à entrada E-, quando CAEX=1, o pino CA0 pode estar conectado à entrada E- e CA1 a entrada E+. Repare que, além da inversão das entradas, o *bit* CAEX também inverte a saída do comparador.

A possibilidade de inversão das entradas do comparador permite reduzir o efeito da tensão de *offset* do comparador, garantindo maior precisão na medição.

Na saída do comparador temos ainda um filtro RC que pode ser ativado (*bit* CACTL2:CAF) para evitar que a saída do comparador oscile demasiadamente quando a diferença de tensão nas entradas E+ e E- é muito pequena. A saída filtrada necessita que o sinal permaneça estável por aproximadamente 2μs, para que haja a comutação do seu estado.

Além do comparador propriamente dito, temos ainda um circuito gerador de tensão de referência programável, que pode ser utilizado para fornecer uma tensão analógica fixa e conhecida para um dos pinos de entrada do comparador.

A tensão de referência pode ser de: ¼ da tensão de alimentação, ½ da tensão de alimentação ou um valor fixo de aproximadamente 0,49 Volts (para uma tensão de 3,3 Volts de alimentação). A seleção da tensão é feita pelos *bits* CAREF1 e CAREF0 do registrador CACTL1.

Observe que a tensão de referência pode ser aplicada a qualquer um dos pinos de entrada do comparador. Essa seleção é feita pelo *bit* CACTL1:CARSEL.

A referência de tensão também pode ser desativada quando ambos os *bits* CAREF estão apagados. Isso contribui para reduzir o consumo de corrente do módulo, quando a referência não é utilizada.

Lembre-se de que os pinos utilizados com o comparador analógico devem ser configurados para a função alternativa (PxSELx = 1) !

5.11.1. Facilidades do Comparador +

Como já foi dito, o comparador + inclui duas facilidades adicionais:

1. Multiplexador analógico nas entradas:

   É possível selecionar um entre três pinos (CA0, CA1 ou CA2) para uma entrada do comparador e um entre sete pinos (CA1, CA2, CA3, CA4, CA5, CA6 ou CA7) para a outra entrada. No primeiro caso, a seleção é feita pelos *bits* P2CA4 e P2CA0, enquanto no segundo caso a seleção é feita pelos *bits* P2CA3, P2CA2 e P2CA1, todos eles localizados no registrador CACTL2.

2. Controle para curto-circuitar as entradas:

   Também é possível fazer com que as entradas E+ e E- do comparador sejam curto-circuitadas. Essa funcionalidade é controlada pelo *bit* CASHORT (registrador CACTL2).

5.11.2. Interrupção do Comparador

A saída do comparador analógico aciona o *flag* CAIFG, localizado no registrador CACTL1.

Esse *flag* é setado na borda de subida ou de descida do sinal de saída do comparador analógico (o que pode ser selecionado pelo *bit* CACTL1:CAIES).

Caso os *bits* SR:GIE e CACTL1:CAIE estejam setados, quando o *flag* CAIFG é setado, o programa desvia para o endereço especificado pelo vetor de número 11, conforme se observa na tabela 5-1.

O *flag* CAIFG é apagado automaticamente quando o programa efetua o desvio para a RTI, mas pode ser também apagado por *software*, no caso de as interrupções não serem utilizadas.

5.11.3. Conexões do Comparador

Os sinais do comparador analógico encontram-se disponíveis nos seguintes pinos do microcontrolador:

Sinal	Pino
CA0	P2.3 (família 1xx e 2xx) P1.6 (família 4xx)
CA1	P2.4 (família 1xx e 2xx) P1.7 (família 4xx)
CA2	P2.0 (família 2xx)
CA3	P2.1 (família 2xx)
CA4	P2.2 (família 2xx)
CA5	P2.5 (família 2xx)
CA6	P2.6 (família 2xx)
CA7	P2.7 (família 2xx)
CAOUT	P2.2 (família 1xx) P2.6 (família 4xx)

Tabela 5-25

5.11.4. Registradores do Comparador Analógico

O módulo comparador analógico utiliza os seguintes registradores:

- **CACTL1** e **CACTL2** - controle do comparador.
- **CAPD** - desabilitação dos *buffers* digitais da porta do comparador.
- **PxSEL** - seleção da função alternativa para os pinos do comparador.

5.11.4.1. CACTL1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0059	CACTL1	Leitura Escrita	CAEX	CARSEL	CAREFx		CAON	CAIES	CAIE	CAIFG
		Reset	0	0	0	0	0	0	0	0

CAEX - inversão das entradas do comparador (o que provoca também a inversão da sua saída) (símbolo **CAEX**);

CARSEL - seleção de uma das entradas do comparador em que será aplicada a tensão de referência interna (V_{CAREF}):

Com CAEX = 0:

0 - referência aplicada à entrada não-inversora;

1 - referência aplicada à entrada inversora (símbolo **CARSEL**);

Com CAEX = 1:

0 - referência aplicada à entrada inversora;

1 - referência aplicada à entrada não-inversora (símbolo **CARSEL**).

CAREFx - seleção da tensão de referência interna:

00 - referência interna desligada. Permite o uso de uma referência externa (símbolo **CAREF_0**);

01 - 0,25 * V_{CC} (símbolo **CAREF_1**);

10 - 0,5 * V_{CC} (símbolo **CAREF_2**);

11 - referência interna de aproximadamente 0,55 Volts (símbolo **CAREF_3**).

CAON - liga/desliga o comparador analógico. A referência de tensão pode ser ligada/desligada independentemente:

0 - comparador desligado;

1 - comparador ligado (símbolo **CAON**).

CAIES - seleção da borda de interrupção do comparador:

0 - borda de subida;

1 - borda de descida (símbolo **CAIES**).

CAIE - habilitação da interrupção do comparador:

0 - desabilitada;

1 - habilitada (símbolo **CAIE**).

CAIFG - sinalizador de interrupção do comparador analógico:

0 - nenhuma interrupção pendente;

1 - a saída do comparador mudou de estado (símbolo **CAIFG**).

5.11.4.2. CACTL2

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x005A	CACTL2	Leitura	CASHORT *	P2CA4*	P2CA3*	P2CA2*	P2CA1	P2CA0	CAF	CAOUT
		Escrita								-
		Reset	0	0	0	0	0	0	0	0

* Esses *bits* somente estão disponíveis nos modelos 2xx.

CASHORT - curto-circuita as entradas do comparador:

0 - entradas não curto-circuitadas;

1 - entradas curto-circuitadas (símbolo **CASHORT**).

P2CA4 -

P2CA0 - seleção da entrada E+ (CAEX=0) ou E- (CAEX=1):

P2CA4	P2CA0	Entrada
0	0	desconectada
0	1	CA0 (símbolo **P2CA0**)
1	0	CA1 (somente nos modelos 2xx)
1	1	CA2 (somente nos modelos 2xx)

P2CA3 -

P2CA2 -

P2CA1 - seleção da entrada E- (CAEX=0) ou E+ (CAEX=1):

P2CA3	P2CA2	P2CA1	Entrada
0	0	0	desconectada
0	0	1	CA1 (símbolo **P2CA1**)
0	1	0	CA2 (somente nos modelos 2xx)
0	1	1	CA3 (somente nos modelos 2xx)
1	0	0	CA4 (somente nos modelos 2xx)
1	0	1	CA5 (somente nos modelos 2xx)
1	1	0	CA6 (somente nos modelos 2xx)
1	1	1	CA7 (somente nos modelos 2xx)

CAF - filtro passa-baixa de saída:

0 - saída não filtrada;

1 - saída filtrada (símbolo **CAF**).

CAOUT - reflete o estado da saída do comparador analógico (símbolo **CAF**).

5.11.4.3. CAPD

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x005B	CAPD	Leitura	CAPD7	CAPD6	CAPD5	CAPD4	CAPD3	CAPD2	CAPD1	CAPD0
		Escrita								
		Reset	0	0	0	0	0	0	0	0

CAPDx - permite desativar os *buffers* digitais dos pinos da porta 1 que estejam sendo utilizados para função analógica. Cada *bit* controla um pino (0 para o pino P1.0, 1 para o pino P1.1, etc.):

0 - *buffer* digital de entrada está ativo;

1 - *buffer* digital de entrada está desabilitado.

5.12. Amplificador Operacional

Os modelos MSP430FG43x incluem um módulo dotado de três amplificadores operacionais independentes (OA0, OA1 e OA2). A figura 5-37 apresenta um diagrama em blocos simplificado para um amplificador.

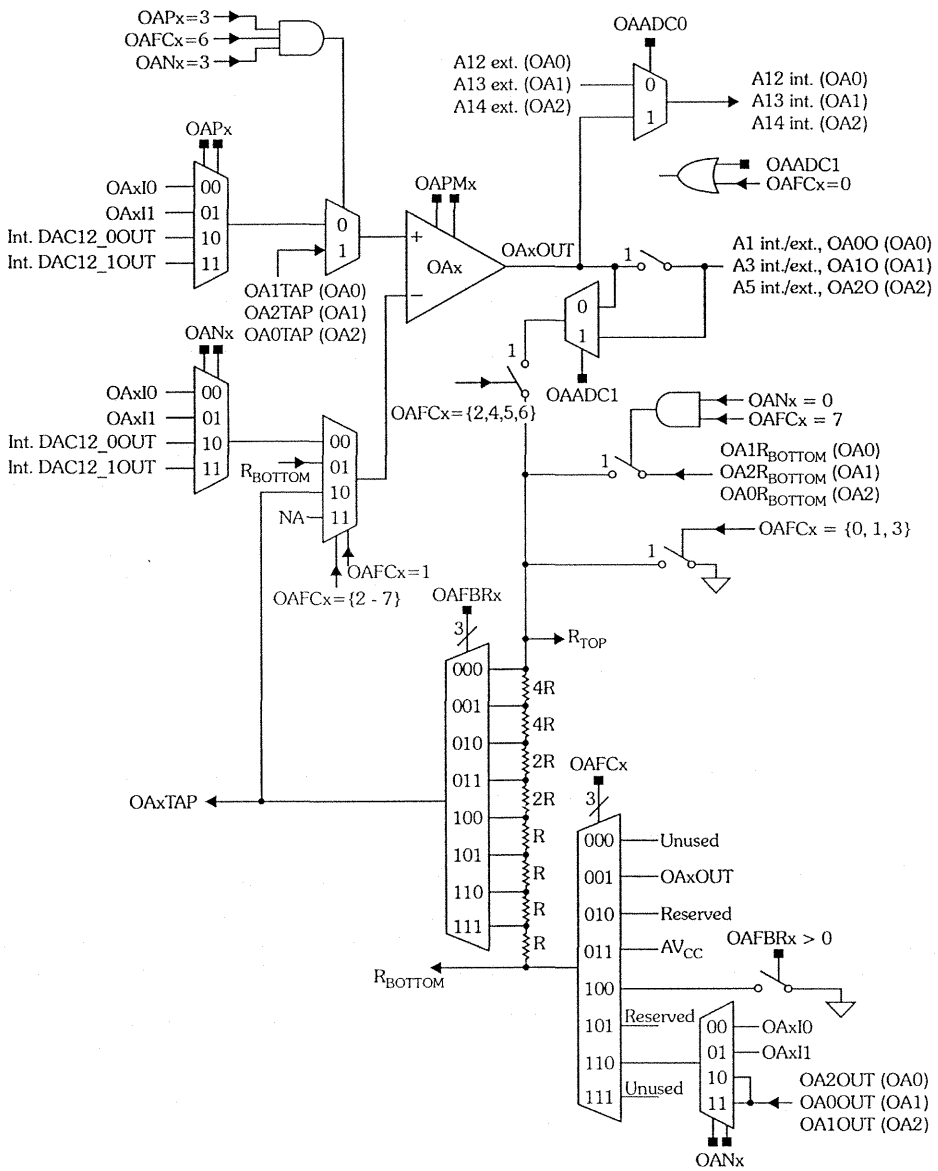


Figura 5-37

As principais características dos amplificadores operacionais são:

- Integração na própria pastilha do MCU, baixo consumo de corrente (médio de 100μA e máximo de 490μA) e tensão de alimentação simples;

- Saída com excursão ampla (*rail-to-rail*);
- Entradas selecionáveis com ampla excursão (*rail-to-rail*);
- Rede de resistores de realimentação configurável por *software*;
- Taxa de variação (*slew rate*) configurável por *software*, de forma a permitir a melhor relação velocidade x consumo;
- Capacidade de operação nos seguintes modos:
  - Amplificadores operacionais independentes;
  - Amplificador de tensão com ganho unitário (*buffer* analógico);
  - Comparador analógico;
  - Amplificador não-inversor com ganho programável;
  - Amplificador inversor com ganho programável;
  - Amplificador diferencial;
- Possibilidade de conexão interna ao ADC12 e DAC12.

O princípio de funcionamento do módulo é relativamente simples: o amplificador operacional pode ter as suas entradas conectadas tanto a pinos externos do microcontrolador (OAxI0 e OAxI1) quanto aos sinais internos provenientes dos outros amplificadores operacionais e do conversor digital/analógico (DAC12). Além disso, cada entrada também pode ser conectada internamente a uma rede de divisores resistivos de tensão, que formam a malha de realimentação do amplificador.

A seleção da conexão da entrada não-inversora do amplificador é feita pelos *bits* OAPx (registrador OAxCTL0). A seleção da conexão da entrada inversora do amplificador é feita pelos *bits* OANx (registrador OAxCTL0).

A malha de realimentação do amplificador é selecionada pelos *bits* OAFBRx, localizados no registrador OAxCTL1.

A saída do amplificador pode ser tanto conectada a uma entrada do conversor analógico/digital (ADC12), ser utilizada para realimentar o amplificador ou ainda ser utilizada para alimentar a entrada de outro amplificador operacional interno. As saídas dos amplificadores não estão disponíveis externamente.

A conexão do sinal de saída do amplificador é controlada pelos *bits* OAADC0 e OAADC1 (registrador OAxCTL0). Quando o *bit* OAADC0 está setado, a saída do amplificador está conectada a entrada A12 (AO0), A13(AO1) ou A14(AO2) do ADC12. Quando o *bit* OAADC1 está setado, a saída do amplificador está conectada à entrada A1(AO0), A3(AO1) ou A5(AO2) do ADC12.

Nestes casos, a respectiva entrada do ADC12 é automaticamente desconectada do pino externo.

5.12.1. Modos de Operação

Os amplificadores operacionais podem funcionar em diversos modos, sendo possível inclusive associá-los para formar elementos mais complexos.

A seleção do modo de operação dos amplificadores operacionais é feita pelos *bits* OAFCx, localizados no registrador OAxCTL1. A tabela 5-26 apresenta os diversos modos de funcionamento para as combinações dos *bits* OAFCx.

OAFCx	Modo
000	Amplificador de uso geral
001	Amplificador de ganho unitário
010	Reservado
011	Comparador analógico
100	Amplificador não-inversor com ganho ajustável
101	Reservado
110	Amplificador inversor com ganho ajustável
111	Amplificador diferencial

Tabela 5-26

5.12.1.1. Amplificador de Uso Geral

Nesse modo, cada amplificador pode ser utilizado de forma independente. As entradas do amplificador são conectadas conforme os *bits* OAxCTL0:OAPx e OAxCTL0:OANx. A saída do amplificador pode ser conectada ao ADC12 conforme o estado do *bit* OAxCTL0:OAADC0.

A malha de realimentação não é conectada ao amplificador.

5.12.1.2. Amplificador de Ganho Unitário

Nesse modo, a saída do amplificador é conectada à malha de realimentação e esta ao pino inversor do amplificador. A seleção dos *bits* OAxCTL1:OAFBRx = 000 permite que se obtenha o ganho unitário.

A entrada não-inversora do amplificador pode ser conectada conforme a seleção dos *bits* OAxCTL0:OAPx. A configuração dos *bits* OAxCTL0:OANx não é considerada, uma vez que a entrada inversora somente está conectada à malha de realimentação.

A saída do amplificador pode ser conectada ao ADC12 conforme o estado dos *bits* OAxCTL0:OAADC1 e OAxCTL0:OAADC0.

5.12.1.3. Comparador

No modo comparador a parte superior (R_{TOP}) da malha de realimentação é conectada ao AV_{CC} e a parte inferior (R_{BOTTOM}) conectada ao AV_{SS}. Neste caso, a seleção do ganho da malha (OAxTAP) permite na realidade selecionar uma tensão de referência para a entrada inversora do amplificador.

A entrada não-inversora é conectada conforme a configuração dos *bits* OAxCTL0:OAPx.

A saída do amplificador pode ser conectada ao ADC12 conforme o estado dos *bits* OAxCTL0:OAADC1 e OAxCTL0:OAADC0.

5.12.1.4. Amplificador Não-Inversor de Ganho Programável

Nesse modo de operação, a saída do amplificador é conectada ao topo da malha de realimentação (R_{TOP}) e a parte inferior da malha é conectada ao terra (AV_{SS}). A seleção da malha de realimentação (OAxTAP) é conectada à entrada inversora do amplificador, permitindo que se selecione a ganho do amplificador pelos *bits* OAxCTL1:OAxFBRx. A tabela 5-27 apresenta os ganhos possíveis.

OAFBRx	Ganho
000	1
001	1,33
010	2
011	2,66
100	4
101	5,33
110	8
111	16

Tabela 5-27

A entrada não-inversora é conectada de acordo com os *bits* OAxCTL0:OAPx. A configuração dos *bits* OAxCTL0:OANx não é considerada, uma vez que a entrada inversora somente está conectada à malha de realimentação.

A saída do amplificador pode ser conectada ao ADC12 conforme o estado dos *bits* OAxCTL0:OAADC1 e OAxCTL0:OAADC0.

5.12.1.5. Amplificador Inversor de Ganho Programável

Nesse modo de operação, a saída do amplificador é conectada ao topo da malha de realimentação (R_{TOP}) e a parte inferior da malha a um multiplexador que permite selecionar entre um dos pinos de entrada do amplificador (OAxI0 ou OAxI1), ou a saída de outro amplificador (o AO0 recebe o sinal do AO1, o AO1 do AO2 e o AO2 do AO0, dependendo do amplificador utilizado). Essa seleção é feita pelos *bits* OAxCTL0:OANx. No caso do amplificador inversor simples, a configuração de OANx deve ser igual a 0 ou 1.

A seleção da malha de realimentação (OAxTAP) é conectada à entrada não-inversora do amplificador, permitindo que se selecione a ganho do amplificador pelos *bits* OAxCTL1:OAxFBRx. A tabela 5-28 apresenta os ganhos possíveis.

OAFBRx	Ganho
000	0
001	-1/3
010	-1
011	-5/3
100	-3
101	-13/3
110	-7
111	-15

Tabela 5-28

A entrada inversora é conectada de acordo com os *bits* OAxCTL0:OANx. A configuração dos *bits* OAxCTL0:OAPx não é considerada, porque a entrada não-inversora somente está conectada à malha de realimentação.

A saída do amplificador pode ser conectada ao ADC12 conforme o estado dos *bits* OAxCTL0:OAADC1 e OAxCTL0:OAADC0.

5.12.1.6. Amplificador Diferencial

É possível implementar um amplificador diferencial utilizando dois ou três amplificadores operacionais internos. A versão com dois operacionais pode ser vista no circuito da figura 5-38.

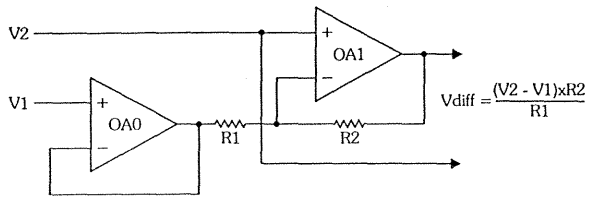


Figura 5-38

As conexões internas para implementar o amplificador podem ser vistas na figura 5-39.

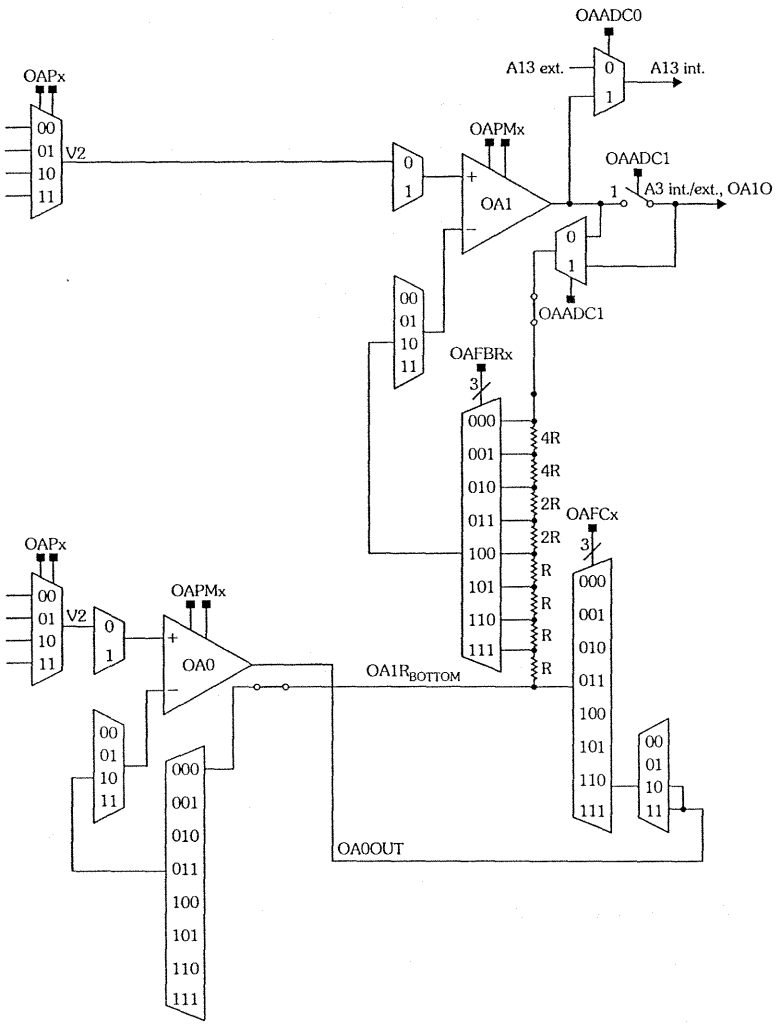


Figura 5-39

A configuração dos registradores do módulo para obter esse circuito é a seguinte:

```
// A seleção de OAFBRx=0 e OAFCx=7 configura o amplificador
// operacional 0 como um amplificador de ganho 1
// A seleção de OANx=0 faz com que a entrada inversora do amplificador
// seja conectada à malha de realimentação do amplificador 1
// A entrada V2 é selecionada pelos bits OAPx do registrador
// OA1CTL0
// A entrada V1 é selecionada pelos bits AOPx do registrador
// OA0CTL0
OA0CTL0 = OAN_0 + OAP_x + OAPM_x; // 00AA BB00
OA0CTL1 = OAFBR_0 + OAFC_7;       // 0001 1100
OA1CTL0 = OAN_2;                  // 10xx xxxx
OA1CTL1 = OAFC_6;                 // xxx1 100x
```

O ganho do amplificador é dado pelo ajuste dos *bits* OAFBRx do registrador OA1CTL1 de acordo com a tabela 5-29.

OAFBRx	Ganho
000	0
001	1/3
010	1
011	5/3
100	3
101	13/3
110	7
111	15

Tabela 5-29

Um amplificador diferencial mais elaborado, utilizando os três amplificadores operacionais internos, pode ser visto na figura 5-40.

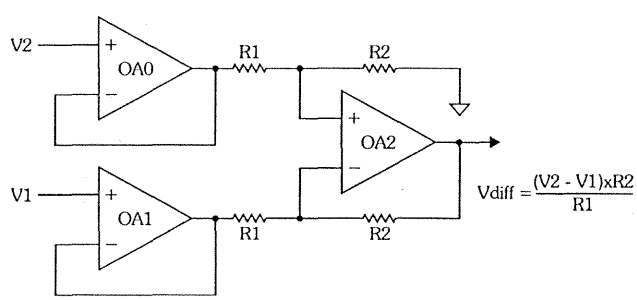


Figura 5-40

O diagrama de conexões dos três amplificadores operacionais internos pode ser visto na figura 5-41.

A configuração dos registradores do módulo para obter este circuito é a seguinte:

```
// A entrada V2 é selecionada pelos bits OAPx do registrador
// OA0CTL0
// A entrada V1 é selecionada pelos bits AOPx do registrador
// OA1CTL0
OA0CTL0 = OAN_0 + OAP_x + OAPM_x; // 00xx xx00
```

```
OA0CTL1 = OAFC_1;                    // xxx0 010x
OA1CTL0 = OAN_0 + OAP_x + OAPM_x;    // 00xx xx00
OA1CTL1 = OAFBR_0 + OAFC_7;          // 0001 110x
OA2CTL0 = OAN_3 + OAP_3              // 1111 xxxx
OA2CTL1 = OAFC_6                     // xxx1 100x
```

O ganho do amplificador é dado pelo ajuste dos *bits* OAFBRx dos registradores OA0CTL1 e OA2CTL1 de acordo com a tabela 5-29. A seleção da malha de realimentação dos dois amplificadores deve ser igual.

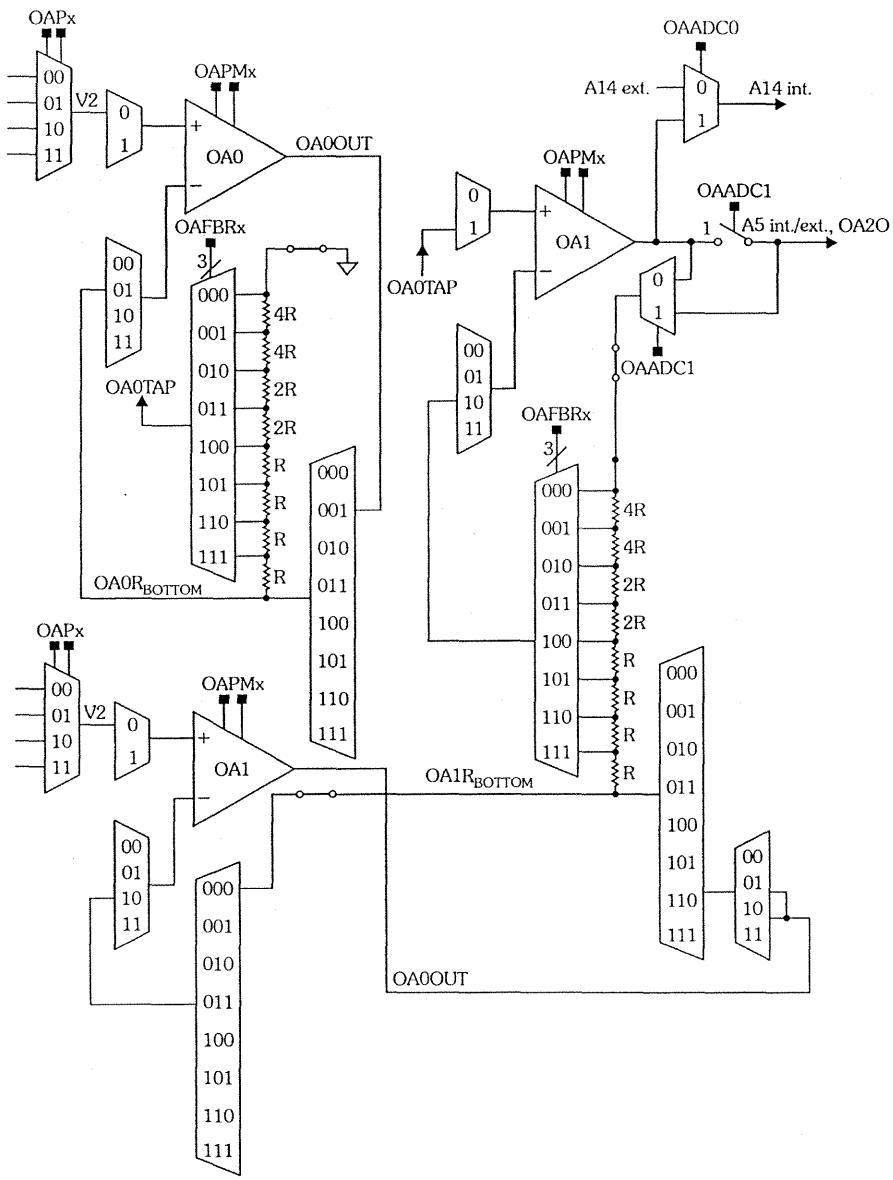


Figura 5-41

5.12.2. Conexões dos Amplificadores Operacionais

Os amplificadores operacionais dispõem de pinos exclusivos para as suas entradas (veja pinagem dos *chips* no tópico 2.5).

5.12.3. Registradores do Amplificador Operacional

Os amplificadores operacionais utilizam os registradores OAxCTL0 e OAxCTL1 para a sua configuração.

5.12.3.1. OAxCTL0

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x00C0	OA0CTL0	Leitura	OANx		OAPx		OAPMx		OAADC1	OAADC0
0x00C2	OA1CTL0	Escrita								
0x00C4	OA2CTL0	*Reset*	0	0	0	0	0	0	0	0

OANx - seleção da entrada inversora do amplificador operacional:
00 - entrada OAxI0 (símbolo **OAN_0**);
01 - entrada OAxI1 (símbolo **OAN_1**);
10 - DAC0 (símbolo **OAN_2**);
11 - DAC1 (símbolo **OAN_3**).

OAPx - seleção da entrada não-inversora do amplificador operacional:
00 - entrada OAxI0 (símbolo **OAP_0**);
01 - entrada OAxI1 (símbolo **OAP_1**);
10 - DAC0 (símbolo **OAP_2**);
11 - DAC1 (símbolo **OAP_3**).

OAPMx - configuração do *slew rate* do amplificador operacional:
00 - desligado, saída em alta impedância (símbolo **OAPM_0**);
01 - lento (símbolo **OAPM_1**);
10 - médio (símbolo **OAPM_2**);
11 - rápido (símbolo **OAPM_3**).

OAADC1 - conexão da saída do amplificador operacional às entradas A1(AO0), A3(AO1) ou A5(AO2):
0 - não conectada;
1 - conectada (símbolo **OAADC1**).

OAADC0 - seleção da saída do amplificador operacional às entradas A12(AO0), A13(AO1) ou A14(AO2):
0 - não conectada;
1 - conectada (símbolo **OAADC0**).

5.12.3.2. OAxCTL1

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x00C1	OA0CTL1	Leitura	OAFBRx			OAFCx			Reservado	OARRIP
0x00C3	OA1CTL1	Escrita								
0x00C5	OA2CTL1	*Reset*	0	0	0	0	0	0	0	0

OAFBRx - seleção do ganho do amplificador operacional (resistor de realimentação):
000 - 0 (símbolo **OAFBR_0**);
001 - 1/3 (símbolo **OAFBR_1**);
010 - 1 (símbolo **OAFBR_2**);
011 - 5/3 (símbolo **OAFBR_3**);
100 - 3 (símbolo **OAFBR_4**);
101 - 13/3 (símbolo **OAFBR_5**);
110 - 7 (símbolo **OAFBR_6**);
111 - 15 (símbolo **OAFBR_7**).

OAFCx - seleção da função do amplificador operacional:
000 - propósito geral (símbolo **OAFC_0**);
001 - *buffer* com ganho unitário (símbolo **OAFC_1**);
010 - reservado (símbolo **OAFC_2**);
011 - comparador de tensão (símbolo **OAFC_3**);
100 - amplificador não-inversor de ganho programável (símbolo **OAFC_4**);
101 - reservado (símbolo **OAFC_5**);
110 - amplificador inversor de ganho programável (símbolo **OAFC_6**);
111 - amplificador diferencial (símbolo **OAFC_7**).

OARRIP - configuração *rail-to-rail* do sinal da entrada:
0 - sinal de entrada é *rail-to-rail*;
1 - o sinal de entrada possui uma excursão limitada (símbolo **OARRIP**).

5.13. Conversor A/D *Slope*

Os MSP430 dotados de comparador analógico permitem implementar um tipo de conversor analógico/digital muito simples, que utiliza o comparador e um canal de captura do *timer* A. Esse tipo de conversor pode ser utilizado para medição de resistências e capacitâncias com grande precisão.

Como podemos observar na figura 5-42, o conversor utiliza o comparador analógico interno para fazer a comparação entre a tensão da entrada CA0 e uma tensão de referência igual a um quarto da tensão de alimentação (0,25 V_{CC}).

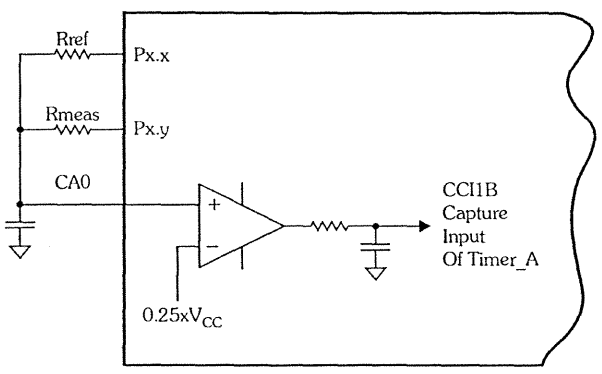


Figura 5-42

O funcionamento do conversor baseia-se na comparação do tempo de descarga do capacitor utilizando uma resistência de referência e a resistência a ser medida. A saída do comparador pode ser ligada à entrada de captura do *timer* A, de forma que cada vez que a tensão no capacitor cai abaixo de 0,25 Vcc, ocorre uma captura no *timer* e com isso podemos medir o tempo de descarga com bastante precisão.

Inicialmente o capacitor é carregado pelos dois resistores (os pinos Px.x e Px.y são configurados como saídas e colocados em nível alto). O tempo de carga deve ser suficiente para que a tensão no capacitor se estabilize próxima a Vcc. Para maior precisão na conversão, podemos utilizar um tempo de carga igual a 7*R*C, sendo R a resistência equivalente do paralelo de Rmeas e Rref em Ohms e C é a capacitância do capacitor em farads.

Uma vez carregado o capacitor, vai ter início a sua descarga pelo resistor de referência (Rref). Antes disso a contagem do *timer* é armazenada. Em seguida, o pino Px.y é configurado como uma entrada. Isso faz com que o resistor Rmeas passe a não influenciar na descarga do capacitor. Então, coloca-se o pino Px.x em nível "0", dando início à descarga do capacitor.

Quando a tensão do capacitor cai abaixo de 0,25 Vcc, a saída do comparador analógico muda de estado, provocando a captura da contagem do *timer* A. A diferença entre o valor dessa captura e a contagem do *timer* no início da descarga é o período de descarga do capacitor (T_{REF}).

Feito isso, o pino Px.y é configurado como uma saída e novamente os dois pinos (Px.x e Px.y) são colocados em nível alto de forma a carregar o capacitor.

Carregado o capacitor, vai ter início a sua descarga pelo resistor de medição (Rmeas). Novamente, a contagem do *timer* é armazenada e em seguida, o pino Px.x é configurado como uma entrada. Isso faz com que o resistor Rref passe a não influenciar na descarga do capacitor. Então, coloca-se o pino Px.y em nível "0", dando início à descarga do capacitor.

Quando a tensão do capacitor cai abaixo de 0,25 Vcc, a saída do comparador analógico muda de estado, provocando a captura da contagem do *timer* A. A diferença entre o valor dessa captura e a contagem do *timer* no início da descarga é o período de descarga do capacitor (T_{MEAS}).

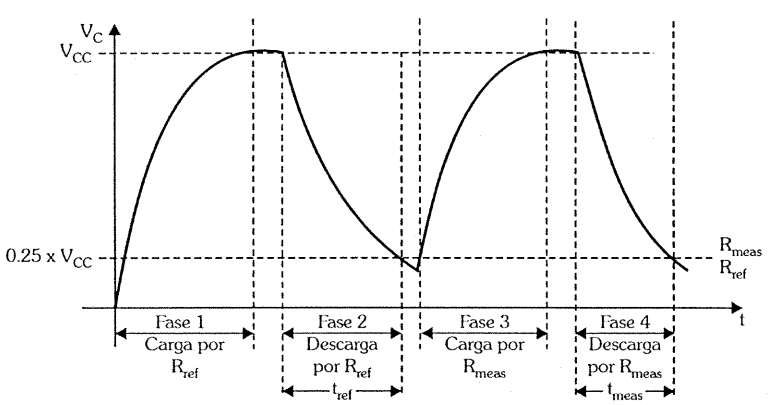


Figura 5-43

Como os períodos de descarga dos dois resistores são conhecidos, a resistência de Rmeas pode ser facilmente calculada pela seguinte fórmula:

$$R_{MEAS} = R_{REF} * \frac{T_{MEAS}}{T_{REF}}$$

5.14. Conversor A/D de 10 *Bits*

Os MSP430F11x2 e 12x2 incluem um conversor analógico/digital de 10 *bits* (ADC10), permitindo ao *chip* a leitura de tensões analógicas.

O conversor possui as seguintes características:

- Resolução de 10 *bits*, monotônico e sem perdas de código;
- Velocidade de até 200.000 amostras por segundo (200Ksps), utilizando a técnica de aproximação sucessiva (SAR);
- Cinco canais (nos modelos 11x2) ou oito canais (modelos 12x2) externos mais quatro internos;
- Operação de 2,2 até 3,6 Volts com consumo típico de aproximadamente 600µA quando operando com tensão de 3 Volts;
- Circuito de amostragem e retenção (*sample-and-hold*) com período de amostragem programável;
- Referência de tensão interna selecionável por *software* (1,5 ou 2,5 Volts) com consumo típico de aproximadamente 250µA quando em operação;
- Referências de tensão externas;
- Canais de entrada exclusivos para sensor de temperatura interno, tensão de alimentação e tensões de referência externas;
- Operação no modo de canal simples, canal simples repetidamente, seqüência de canais ou seqüência de canais repetidamente;
- Controlador de transferência de dados (DTC) que permite a transferência automática do resultado da conversão para a memória do microcontrolador, sem a interferência da CPU;
- Fontes de *clock* selecionáveis por *software*;
- Possibilidade de iniciar uma conversão por uma das saídas do *timer* A.

O funcionamento do módulo consiste basicamente no seguinte: o canal de entrada analógico, selecionado pelo multiplexador analógico, com o uso dos *bits* ADC10CTL1:INCHx é conectado ao circuito de amostragem e retenção (*sample-and-hold*), que consiste basicamente em uma chave analógica seguida de um capacitor.

O circuito de amostragem e retenção é comandado pelo temporizador de amostragem que tem a função de iniciar e temporizar a fase de amostragem do sinal analógico. A duração do ciclo de amostragem é controlada pelos *bits* ADC10SHTx (registrador ADC10CTL0) e pode ser de 4, 8, 16 ou 64 ciclos de *clock* do ADC10.

O período mínimo de amostragem do sinal de entrada é definido pela resistência interna da fonte do sinal e pode ser determinado pela seguinte equação:

$$t_{amostragem} = (R_S + 2000) * 7{,}625 * 20 * 10^{-12} + 800 * 10^{-9}$$

sendo:

- $t_{amostragem}$ - tempo mínimo de amostragem em segundos
- R_S - resistência interna da fonte de sinal em Ohms

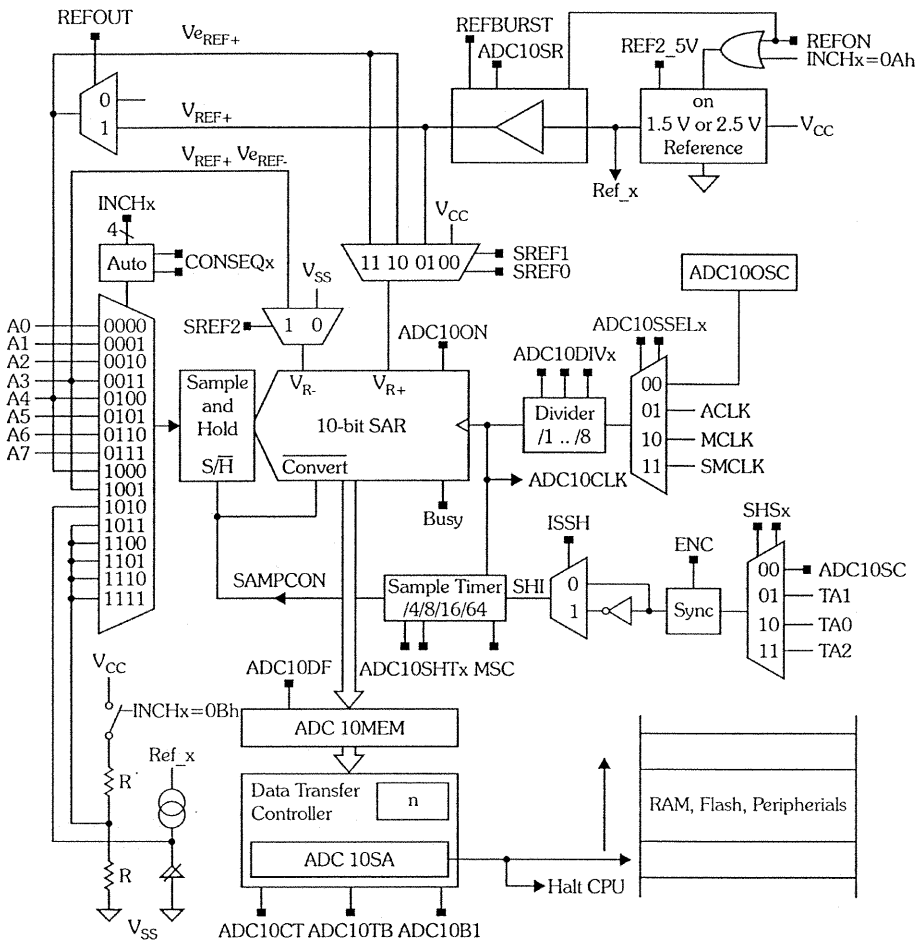


Figura 5-44

Quando opera no modo de alta velocidade (*bit* ADC10CTL0:ADC10SR=1), é necessário adicionar um tempo de 1,7µs ao tempo calculado anteriormente.

Após a fase de amostragem, tem início a conversão do sinal pelo circuito SAR. São necessários 13 ciclos de *clock* (ADC10CLK) para que ela seja concluída.

A fonte de *clock* do conversor pode ser selecionada pelos *bits* ADC10SSELx (registrador ADC10CTL1), conforme a tabela:

ADC10SSELx	Fonte de *clock*
00	Oscilador interno do ADC10
01	ACLK
10	MCLK
11	SMCLK

Tabela 5-30

O oscilador interno do ADC10 opera a aproximadamente 5MHz (a margem média de tolerância é de 3,6 a 6,3 MHz).

A freqüência de *clock* selecionada pode ser dividida por um fator entre 1 e 8, conforme a seleção dos *bits* ADC10DIVx (registrador ADC10CTL1).

Muito cuidado deve ser tomado para que o conversor opere com uma freqüência de *clock* entre 450KHz e 6,3MHz. A operação fora desta faixa não é garantida pelo fabricante e pode estar sujeita a erros e perda de linearidade.

O disparo de uma amostragem e conversão pode ser feito por uma das quatro fontes selecionadas pelos *bits* SHSx (registrador ADC10CTL1):

SHSx	Fonte de disparo
00	*software* (*bit* ADC10CTL0:ADC10SC)
01	*timer* A (saída TA1)
10	*timer* A (saída TA0)
11	*timer* A (saída TA2)

Tabela 5-31

O disparo por *software* é comandado pelo *bit* ADC10SC (registrador ADC10CTL0) que é automaticamente apagado pelo *hardware* após o início da amostragem/conversão.

As demais fontes são todas provenientes do *timer* A, o que permite que uma conversão seja iniciada por um evento de comparação em um dos canais do *timer*.

A polaridade do sinal de disparo pode ser selecionada pelo *bit* ISSH (registrador ADC10CTL1).

Repare que, em todos os casos, uma amostragem/conversão somente será iniciada se o conversor estiver ligado (ADC10CTL0:ADC10ON = 1), a fonte de *clock* selecionada estiver ativa e as conversões estiverem habilitadas (ADC10CTL0:ENC = 1).

Durante um ciclo de amostragem/conversão, o *flag* ADC10BUSY (registrador ADC10CTL1) permanece setado, sendo apagado automaticamente após o término da conversão.

Completada a conversão, o resultado é armazenado no registrador ADC10MEM ou copiado para a memória pelo controlador DTC. Maiores detalhes sobre o controlador DTC serão vistos mais adiante.

O formato do resultado da conversão é determinado pelo *bit* ADC10DF (registrador ADC10CTL1): com ele em "0" o resultado é armazenado no formato binário, justificado à direita (os seis *bits* mais significativos do destino são mantidos em zero). Quando ADC10DF = 1, o resultado é armazenado no formato complemento de dois, justificado à esquerda (os seis *bits* menos significativos do destino são mantidos em zero).

Caso um modo de repetição ou de seqüência esteja selecionado e o *bit* MSC (registrador ADC10CTL0) esteja setado, uma nova amostragem/conversão é iniciada logo após a anterior ser completada.

Os modos de seqüência e repetição permitem que o conversor efetue a leitura de um ou mais canais, repetidamente ou a cada novo disparo de conversão. A seleção dos modos é feita pelos *bits* CONSEQx (registrador ADC10CTL1), conforme a tabela seguinte:

CONSEQx	MSC	Modo	Descrição
00	x	Um canal sem repetição	O canal selecionado pelos *bits* ADC10CTL1:INCHx é amostrado e convertido. O resultado é escrito no registrador ADC10MEM. Cada nova conversão necessita de um novo disparo. No caso de a fonte de disparo ser um dos canais do *timer* A, também é necessário que o *bit* ENC seja apagado e novamente setado antes da próxima conversão. O estado do *bit* MSC nesse modo é indiferente.
01	0	Seqüência de canais sem repetição	Uma seqüência de canais que vai daquele selecionado pelos *bits* ADC10CTL1:INCHx até o canal A0 é convertida uma única vez. Cada resultado é armazenado no registrador ADC10MEM. Uma nova conversão da seqüência necessita de um novo disparo. No caso de a fonte de disparo ser um dos canais do *timer* A, também é necessário que o *bit* ENC seja apagado e novamente setado antes da próxima conversão.
10	1	Um canal repetidamente	O canal selecionado pelos *bits* ADC10CTL1:INCHx é amostrado e convertido continuamente. Cada resultado é escrito no registrador ADC10MEM. Para encerrar as conversões, basta apagar o *bit* ENC.
11	1	Seqüência de canais repetidamente	Uma seqüência de canais que vai daquele selecionado pelos *bits* ADC10CTL1:INCHx até o canal A0 é convertida repetidamente. Cada resultado é armazenado no registrador ADC10MEM. Para encerrar as conversões, basta apagar o *bit* ENC.

Tabela 5-32

Uma facilidade adicional disponível no módulo ADC10 é a possibilidade de desligar os *buffers* digitais dos pinos de entrada do conversor A/D, de forma que as correntes parasitas sejam minimizadas, aumentando a precisão do conversor.

O registrador ADC10AE permite que se desabilitem os pinos da porta 2 (e 3 no caso dos MSP430F12x2), utilizados como entradas e saídas do conversor.

5.14.1. Referências de Tensão

O ADC10 pode operar utilizando tanto uma das suas referências internas quanto uma ou duas referências externas. As entradas de referência V_{R+} e V_{R-} permitem definir os limites superior e inferior da conversão, ou seja, o maior e o menor valor de tensão que o sistema pode converter.

Isso significa que, quando a tensão de entrada é igual ou maior que V_{R+}, a saída do conversor é igual a 1023 (0x03FF) e quando a tensão de entrada é menor ou igual a V_{R-}, a saída do conversor é igual a 0.

Podemos selecionar a referência a ser utilizada pelos *bits* SREFx do registrador ADC10CTL0, conforme a tabela seguinte:

SREFx	V_{R+}	V_{R-}
000	V_{CC}	V_{SS}
001	V_{REF+}	V_{SS}
010	Ve_{REF+}	V_{SS}
011	Ve_{REF+}	V_{SS}
100	V_{CC}	V_{REF-}/Ve_{REF-}
101	V_{REF+}	V_{REF-}/Ve_{REF-}
110	Ve_{REF+}	V_{REF-}/Ve_{REF-}
111	Ve_{REF+}	V_{REF-}/Ve_{REF-}

Tabela 5-33

A opção V_{REF+} é proveniente do gerador de tensão de referência interno e pode ser ativada e desativada pelo *bit* REFON (registrador ADC10CTL0). A tensão de referência, uma vez ativada, pode ser selecionada entre dois valores: 1,5 ou 2,5 Volts. A seleção é feita por meio do *bit* REF25V (registrador ADC10CTL0).

A tensão selecionada tem a sua corrente amplificada por um *buffer* analógico. É possível selecionar a corrente máxima de saída do *buffer* e com isso fazer a opção entre alta corrente (permitindo velocidades de conversão de até 200Ksps) ou baixa corrente (o que limita a velocidade de conversão a 50Ksps). A seleção é realizada pelo *bit* ADC10SR, localizado no registrador ADC10CTL0.

A tensão de referência interna pode ainda ser disponibilizada externamente, desde que o *bit* REFOUT (registrador ADC10CTL0) esteja setado. Observe, no entanto, que essa prática aumenta o consumo de corrente do módulo em cerca de 0,5 a 1mA, conforme o dispositivo.

Além da tensão de referência interna, podemos utilizar uma tensão de referência externa (Ve_{REF+}) aplicada à entrada A4 do conversor, além de outra para a margem inferior (V_{REF-}/Ve_{REF-}) que pode ser aplicada à entrada A3. Alternativamente, podemos também utilizar como referência a própria tensão de alimentação do chip (AV_{CC} e AV_{SS}).

O resultado da conversão pode ser expresso pela seguinte fórmula:

$$R_{ADC} = 1023 * \frac{V_{in} - V_{R-}}{V_{R+} - V_{R-}}$$

5.14.2. Controlador de Transferência de Dados

Os microcontroladores MSP430 dotados de conversor A/D de 10 *bits* implementam um circuito que pode ser utilizado para a transferência automática do resultado da conversão para a memória RAM, minimizando a utilização da CPU.

O controlador (DTC) pode ser programado para transferir "n" resultados para um bloco localizado a partir de um endereço específico da memória RAM.

Cada transferência tem início após o término de uma conversão e a escrita do resultado no registrador ADC10MEM, necessitando de um ciclo MCLK para ser completada. Durante esse tempo, a CPU é mantida paralisada (a execução do programa é momentaneamente suspensa).

Após a transferência dos "n" resultados de conversão, o *flag* ADC10CTL0:ADC10IFG é setado e as transferências são encerradas. Também é possível fazer com que o DTC opere continuamente, reiniciando as transferências após a conclusão de um bloco. Basta setar o *bit* ADC10CT (registrador ADC10DTC0).

O número de transferências ("n") a serem realizadas é determinado pelo conteúdo do registrador ADC10DTC1. O limite máximo é de 255 transferências, o que restringe um bloco ao tamanho máximo de 255 *words* de memória.

O endereço inicial do bloco é indicado pelo registrador ADC10SA e pode estar localizado em qualquer endereço par dentro do espaço de 64K endereços disponíveis. Normalmente, o bloco se localiza na área de RAM do dispositivo. As transferências somente têm início após a escrita nesse registrador.

A cada operação de transferência um registrador de endereços interno (inacessível ao usuário) é incrementado em dois e o contador de transferências é decrementado de um.

Além do modo de operação descrito anteriormente, o módulo ADC10 pode também operar no modo de transferência de dois blocos (setando o *bit* ADC10TB no registrador ADC10DTC0).

No modo de dois blocos, o DTC realiza a transferência de dois blocos seqüenciais: o primeiro localizado do endereço indicado por ADC10SA até o endereço ADC10SA+2*n-2. O segundo bloco inicia-se imediatamente após o primeiro e vai do endereço ADC10SA+2*n até o endereço ADC10SA+4*n.

A cada transferência de bloco completada, o *flag* ADC10CTL0:ADC10IFG é setado. Além disso, um *bit* adicional chamado ADC10DTC0:ADC10B1 pode ser utilizado para identificar qual dos blocos teve a sua transferência completada: ADC10B1 = 1 indica que o bloco completado foi o primeiro; caso contrário, trata-se do segundo bloco.

Caso o *bit* ADC10CT (registrador ADC10DTC0) esteja setado, após a transferência do segundo bloco, ao término da próxima conversão o DTC reinicia a operação de transferência a partir do endereço inicial do primeiro bloco.

Observe que, setado o ADC10CT, o seu apagamento não implica no encerramento imediato das operações do DTC. No caso do modo de um bloco, o DTC continua as operações de transferência até o final daquele bloco e no modo de dois blocos, o DTC somente encerra a operação após o término do segundo bloco.

5.14.3. Configuração

A configuração do ADC10, de maneira geral, deve observar os seguintes passos:

1. O *bit* ADC10CTL0:ENC deve ser apagado.
2. Selecionam-se a fonte de *clock*, fator de divisão dele, tempo de amostragem, referência de tensão, canal de entrada, modo de conversão e fonte de disparo da conversão (registradores ADC10CTL0 e ADC10CTL1).
3. Habilita-se o ADC10 (ADC10CTL0:ENC = 1).
4. O(s) canal(is) de entrada deve(m) ser habilitado(s) no registrador ADC10AE.
5. Caso desejado, habilita-se a interrupção do módulo (ADC10CTL0:ADC10IE = 1).

5.14.4. Sensor de Temperatura

Os microcontroladores MSP430 dotados de conversor A/D de 10 *bits* incluem um sensor de temperatura integrado à pastilha do *chip*.

A utilização do sensor é muito simples. Basta selecionar o canal de entrada número 10. Com ele selecionado, o sensor de temperatura é automaticamente ativado, assim como a referência interna de temperatura.

O restante da configuração do ADC10 deve ser feita normalmente, como no caso da utilização de uma entrada externa.

A função típica de transferência do sensor pode ser expressa pela seguinte equação:

$$\text{TEMP}(^{\circ}\text{C}) = \frac{V_{\text{TEMP}} - 0{,}986}{0{,}00355}$$

Repare que para a correta utilização do sensor de temperatura é importante utilizar um tempo de amostragem superior a 30µs.

Além disso, os valores medidos podem apresentar um *offset* que varia conforme o dispositivo, e pode ser necessária uma etapa de calibração antes de usar o sensor.

5.14.5. Interrupções

O ADC10 dispõe de um vetor de interrupção exclusivo localizado no endereço 0xFFEA (vetor de número 5).

O *flag* ADC10IFG (registrador ADC10CTL0) é setado de acordo com o modo de operação do DTC.

Caso o DTC não esteja sendo utilizado (registrador ADC10DTC1 = 0), o *flag* ADC10IFG é setado sempre que um novo resultado de conversão tenha sido escrito no registrador ADC10MEM.

No caso de utilização do DTC, o *flag* ADC10IFG é setado sempre que a transferência de um bloco inteiro tiver sido completada.

O *flag* ADC10IFG é automaticamente apagado após o desvio para a rotina de tratamento de interrupção (o que acontece quando o SR:GIE e ADC10CTL0:ADC10IE estão ambos setados). Além disso, ele também pode ser apagado pelo *software* em execução.

5.14.6. Conexões do ADC10

O ADC10 pode utilizar os seguintes pinos:

Sinal	Pino
A0	P2.0
A1	P2.1
A2	P2.2
A3 / V_{REF-} / Ve_{REF-}	P2.3
A4 / V_{REF+} / Ve_{REF+}	P2.4
A5	P3.0 (somente nos modelos 12x2)
A6	P3.6 (somente nos modelos 12x2)
A7	P3.7 (somente nos modelos 12x2)
ADC10CLK	P1.0

Tabela 5-34

O pino P1.0 pode ser configurado para disponibilizar externamente o sinal de *clock* do conversor (ADC10CLK). Para isso é necessário configurar o pino como saída (P1DIR.0 = 1) e para a função alternativa (P1SEL.0 = 1).

5.14.7. Registradores

O ADC10 utiliza os seguintes registradores para a sua configuração e operação:

- **ADC10CTL0** e **ADC10CTL1** - configuração do ADC10;
- **ADC10MEM** - resultado da conversão;
- **ADC10DTC0** - configuração do DTC;

- **ADC10DTC1** - tamanho do bloco a ser transferido;
- **ADC10SA** - endereço inicial do bloco;
- **ADC10AE** - habilitação das entradas analógicas.

5.14.7.1. ADC10CTL0

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01B0	ADC10CTL0	Leitura / Escrita	SREFx*			ADC10SHTx*		ADC10SR*	REFOUT*	REF BURST*
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura / Escrita	MSC*	REF2_5V*	REFON*	ADC10ON*	ADC10IE	ADC10IFG	ENC	ADC10SC
		Reset	0	0	0	0	0	0	0	0

* Esses *bits* somente podem ser alterados com ENC = 0.

SREFx - seleção das referências do conversor:

SREFx	V_{R+}	V_{R-}	Símbolo
000	V_{CC}	V_{SS}	**SREF_0**
001	V_{REF+}	V_{SS}	**SREF_1**
010	Ve_{REF+}	V_{SS}	**SREF_2**
011	Ve_{REF+}	V_{SS}	**SREF_3**
100	V_{CC}	V_{REF-}/Ve_{REF-}	**SREF_4**
101	V_{REF+}	V_{REF-}/Ve_{REF-}	**SREF_5**
110	Ve_{REF+}	V_{REF-}/Ve_{REF-}	**SREF_6**
111	Ve_{REF+}	V_{REF-}/Ve_{REF-}	**SREF_7**

ADC10SHTx - seleção do tempo de amostragem:
00 - 4 ciclos do ADC10CLK (símbolo **ADC10SHT_0**);
01 - 8 ciclos do ADC10CLK (símbolo **ADC10SHT_1**);
10 - 16 ciclos do ADC10CLK (símbolo **ADC10SHT_2**);
11 - 64 ciclos do ADC10CLK (símbolo **ADC10SHT_3**).

ADC10SR - corrente de saída do buffer de referência (afeta diretamente o consumo e a velocidade de amostragem:
0 - velocidade de até 200ksps (maior consumo de corrente);
1 - velocidade de até 50ksps (menor consumo de corrente)(símbolo **ADC10SR**).

REFOUT - ativação da saída de referência externa:
0 - saída de referência desligada;
1 - saída de referência ligada (símbolo **REFOUT**).

REFBURST - controle da saída de referência externa:
0 - saída de referência opera continuamente;
1 - saída de referência opera somente durante uma amostragem/conversão (símbolo **REFBURST**).

MSC - seleção de múltiplas conversões (somente nos modos de seqüência ou repetitivos):

0 - modo de conversão única (um evento de disparo de conversão provoca somente uma conversão);

1 - modo de conversão repetitiva (um evento de disparo provoca o início de múltiplas conversões) (símbolo **MSC**).

REF2_5V - seleção da tensão de saída do gerador de tensão de referência (somente quando REFON = 1, ou seja, quando o gerador de referência estiver ligado):

0 - 1,5 Volts;

1 - 2,5 Volts (símbolo **REF2_5V**).

REFON - ativação do gerador de tensão de referência interno:

0 - desligado;

1 - ligado (símbolo **REFON**).

ADC10ON - controle liga/desliga do módulo ADC10:

0 - desligado;

1 - ligado (símbolo **ADC10ON**).

ADC10IE - controle de habilitação da interrupção do conversor:

0 - desabilitada;

1 - habilitada (símbolo **ADC10IE**).

ADC10IFG - sinalizador de interrupção do ADC10. É setado após a escrita do resultado da conversão no registrador ADC10MEM ou após a transferência do bloco para a memória (quando utilizando o DTC). Esse *flag* é apagado automaticamente pelo *hardware* quando ocorre a chamada da rotina de tratamento de interrupção, podendo também ser apagado por *software*:

0 - nenhuma interrupção;

1 - interrupção pendente (símbolo **ADC10IFG**).

ENC - habilitação da conversão:

0 - conversor desabilitado;

1 - conversor habilitado (símbolo **ENC**).

ADC10SC - início da conversão. Esse *bit*, quando setado, provoca o início de uma conversão (desde que a fonte de disparo selecionada seja o *software*). O *bit* é automaticamente apagado pelo *hardware* após o início da amostragem:

0 - nenhuma conversão;

1 - iniciar amostragem/conversão (símbolo **ADC10SC**).

5.14.7.2. ADC10CTL1

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01B2	ADC10CTL1	Leitura / Escrita	INCHx*				SHSx*		ADC10DF*	ISSH*
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura / Escrita	ADC10DIVx*			ADC10SSELx*		CONSEQx*		ADC10 BUSY
		Reset	0	0	0	0	0	0	0	0

* Esses *bits* somente podem ser alterados com ENC = 0.

INCHx - seleção do canal de entrada:

INCHx	Canal	Símbolo
0000	A0	**INCH_0**
0001	A1	**INCH_1**
0010	A2	**INCH_2**
0011	A3	**INCH_3**
0100	A4	**INCH_4**
0101	A5	**INCH_5**
0110	A6	**INCH_6**
0111	A7	**INCH_7**
1000	Ve_{REF+}	**INCH_8**
1001	V_{REF-} / Ve_{REF-}	**INCH_9**
1010	Diodo de temperatura	**INCH_10**
1011	$(V_{CC}-V_{SS})/2$	**INCH_11**
1100	$(V_{CC}-V_{SS})/2$	**INCH_12**
1101	$(V_{CC}-V_{SS})/2$	**INCH_13**
1110	$(V_{CC}-V_{SS})/2$	**INCH_14**
1111	$(V_{CC}-V_{SS})/2$	**INCH_15**

SHSx - seleção do disparo do conversor:

SHSx	Fonte de disparo	Símbolo
00	*software* (*bit* ADC10CTL0:ADC10SC)	**SHS_0**
01	*timer* A (saída TA1)	**SHS_1**
10	*timer* A (saída TA0)	**SHS_2**
11	*timer* A (saída TA2)	**SHS_3**

ADC10DF - seleção do formato de saída do ADC10:
0 - binário justificado à direita;
1 - complemento de dois, justificado à esquerda (símbolo **ADC10DF**).

ISSH - inversão do sinal início de amostragem e conversão:
0 - o sinal de disparo não é invertido;
1 - o sinal de disparo é invertido (símbolo **ISSH**).

ADC10DIV - seleção do divisor do *clock* do ADC10:

ADC10DIV	Fator de divisão	Símbolo
000	1	**ADC10DIV_0**
001	2	**ADC10DIV_1**
010	3	**ADC10DIV_2**
011	4	**ADC10DIV_3**
100	5	**ADC10DIV_4**
101	6	**ADC10DIV_5**
110	7	**ADC10DIV_6**
111	8	**ADC10DIV_7**

ADC10SSELx - seleção da fonte de *clock* do ADC10:

ADC10SSELx	Fonte de *clock*	Símbolo
00	ADC10OSC	**ADC10SSEL_0**
01	ACLK	**ADC10SSEL_1**
10	MCLK	**ADC10SSEL_2**
11	SMCLK	**ADC10SSEL_3**

CONSEQx - inversão do sinal início de amostragem e conversão:

CONSEQx	Modo de conversão	Símbolo
00	Um canal, conversão simples	**CONSEQ_0**
01	Seqüência de canais	**CONSEQ_1**
10	Um canal, repetitivo	**CONSEQ_2**
11	Seqüência de canais, repetitivo	**CONSEQ_3**

ADC10BUSY - indicador de atividade do ADC10 (somente leitura):
0 - nenhuma operação em andamento;
1 - uma operação de amostragem ou conversão está em andamento (símbolo **ADC10BUSY**).

5.14.7.3. ADC10AE

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x004A	ADC10AE	Leitura / Escrita	ADC10AE 7	ADC10AE 6	ADC10AE 5	ADC10AE 4	ADC10AE 3	ADC10AE 2	ADC10AE 1	ADC10AE 0
		Reset	0	0	0	0	0	0	0	0

ADC10AEx - habilita ou desabilita os *buffers* digitais das entradas do ADC10:
0 - entrada analógica desabilitada;
1 - entrada analógica habilitada.

5.14.7.4. ADC10MEM

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01B4	ADC10MEM	Leitura / Escrita	Resultado 16 *bits* da conversão							
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura / Escrita	Resultado 16 *bits* da conversão							
		Reset	0	0	0	0	0	0	0	0

O registrador ADC10MEM armazena o resultado da conversão A/D. Quando o *bit* ADC10CTL1:ADC10DF = 0, o resultado é armazenado nos 10 *bits* menos significativos e os *bits* 10 a 15 são mantidos em "0".

Quando ADC10DF = 1, o resultado é armazenado no formato complemento de dois, alinhado à esquerda. O MSB do resultado é armazenado no *bit* 15, o LSB é armazenado no *bit* 6, os *bits* 5 até 0 são mantidos em "0".

5.14.7.5. ADC10DTC0

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0048	ADC10DTC0	Leitura	Reservados				ADC10TB	ADC10CT	ADC10B1	ADC10 FETCH
		Escrita	-							
		Reset	0	0	0	0	0	0	0	0

ADC10TB - seleção do modo de transferência de dois blocos:

0 - modo de um bloco;

1 - modo de dois blocos (símbolo **ADC10TB**).

ADC10CT - modo de transferência contínua:

0 - a transferência é encerrada após completado um bloco (ADC10TB=0) ou dois blocos (ADC10TB=1);

1 - a transferência é contínua. Para encerrar o processo, basta apagar o *bit* ADC10CT ou escrever um novo valor no ADC10SA (símbolo **ADC10CT**).

ADC10B1 - indicador do primeiro/segundo bloco:

0 - o bloco 2 foi o último completado;

1 - o bloco 1 foi o último completado (símbolo **ADC10B1**).

ADC10FETCH - esse *bit* deve ser mantido em nível "0"

5.14.7.6. ADC10DTC1

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0049	ADC10DTC1	Leitura	Número de transferências por bloco: 0 = nenhuma							
		Escrita								
		Reset	0	0	0	0	0	0	0	0

5.14.7.7. ADC10SA

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01BC	ADC10SA	Leitura	Endereço inicial do bloco							
		Escrita								
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	Endereço inicial do bloco							0
		Escrita								
		Reset	0	0	0	0	0	0	0	0

A escrita no registrador ADC10SA é necessária para iniciar as transferências do DTC.

5.15. Conversor A/D de 12 *Bits*

Os modelos MSP430F13x, 14x, 15x, 16x, F43x, FG437 e F44x possuem um conversor A/D interno de 12 *bits* (ADC12).

As características de funcionamento do ADC12 são bastante parecidas com as do ADC10, no entanto o ADC12 apresenta algumas particularidades:

- Resolução de 12 *bits*, monotônico e sem perdas de código;
- Velocidade de até 200.000 amostras por segundo (200Ksps), utilizando a técnica de aproximação sucessiva (SAR);
- Oito canais externos (12 nos dispositivos MSP430FG43x) mais quatro internos;
- Operação de 2,2 até 3,6 Volts, com consumo típico de aproximadamente 800μA quando operando com tensão de 3 Volts;
- Circuito de amostragem e retenção (*sample-and-hold*) com período de amostragem programável e modo de amostragem estendida;
- Referência de tensão interna selecionável por *software* (1,5 ou 2,5 Volts) com consumo típico de aproximadamente 250μA quando em operação;
- Referências de tensão externas;
- Canais de entrada exclusivos para sensor de temperatura interno, tensão de alimentação e tensões de referência externas;
- Operação no modo de canal simples, canal simples repetidamente, seqüência de canais ou seqüência de canais repetidamente;
- Fontes de *clock* selecionáveis por *software*;
- Possibilidade de iniciar uma conversão por uma das saídas dos *timers* A e B;
- 16 memórias de conversão com controle independente de cada uma, inclusive com a capacidade de especificar o canal de entrada e a referência;
- 18 fontes de interrupção, sendo 16 para os canais e mais duas para erros do conversor;
- O ADC12 não dispõe de controlador de transferência de dados (em vez disso utiliza-se o DMA, quando estiver disponível.

O funcionamento do módulo consiste basicamente no seguinte:

1. Seleciona-se o modo de funcionamento do conversor (conversão simples ou seqüência de canais), pelos *bits* CONSEQx (registrador ADC12CTL1).
2. Seleciona-se o endereço inicial da memória de conversão, pelos *bits* CSTARTADDx (registrador ADC12CTL1).
3. Liga-se o conversor (*bit* ADC12CTL0:ADC12ON=1) e habilitam-se as conversões (*bit* ADC12CTL0:ENC=1).

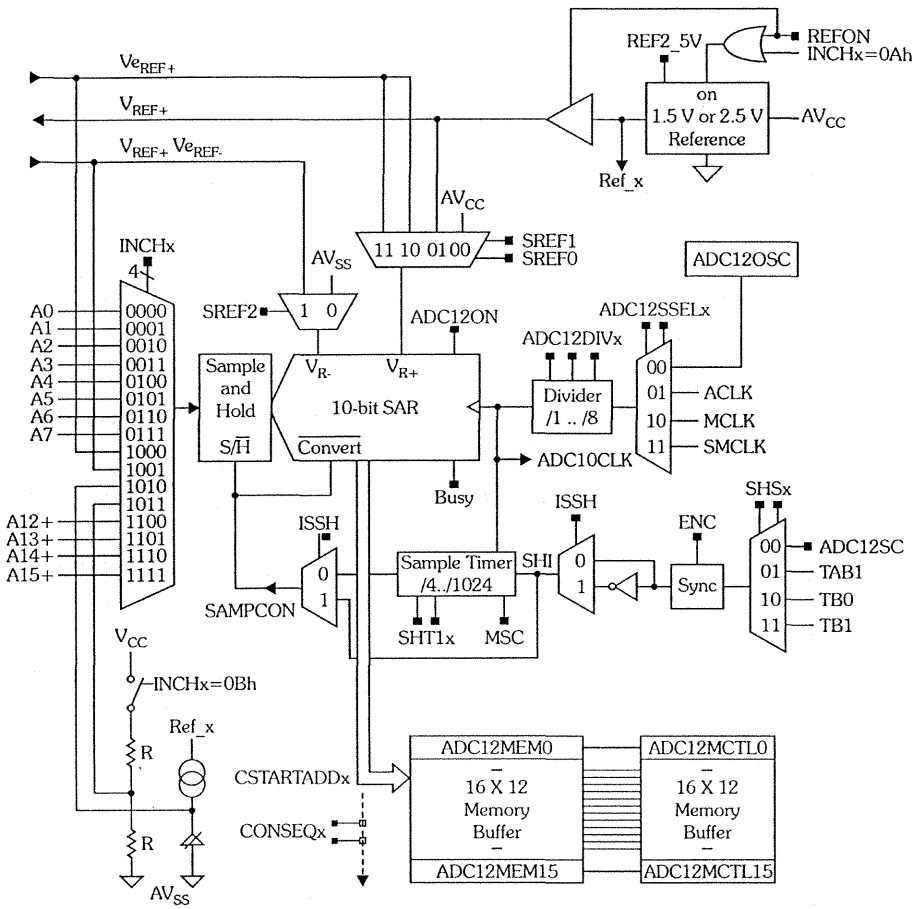


Figura 5-45

O conversor aguarda a ocorrência de um sinal de disparo de conversão (SAMPCON na figura 5-45) que pode ser originado de uma das quatro fontes selecionadas pelos *bits* SHSx (registrador ADC12CTL1):

SHSx	Fonte de disparo
00	*software* (*bit* ADC12CTL0:ADC12SC)
01	*timer* A (saída TA1)
10	*timer* B (saída TB0)
11	*timer* B (saída TB1)

Tabela 5-35

O disparo por *software* é comandado pelo *bit* ADC12SC (registrador ADC12CTL0) que é automaticamente apagado pelo *hardware* após o início da amostragem/conversão.

As demais fontes são provenientes dos *timers* A e B, o que permite que uma conversão seja iniciada por um evento de comparação em um dos canais dos *timers*.

A polaridade do sinal de disparo pode ser selecionada pelo *bit* ISSH (registrador ADC12CTL1).

Um aspecto importante e diferenciado do ADC12 é que o modo como é feita a amostragem pode ser controlado, pelo *bit* SHP (registrador ADC12CTL1).

Com SHP = 0 temos o modo de amostragem estendida, no qual o período de amostragem é determinado pela duração do sinal interno SHI e este por sua vez é comandado pelo sinal de disparo de conversão, selecionado conforme a tabela 5-35 anterior. Quanto maior a duração do pulso de disparo maior será o período de amostragem.

Esse modo pode ser particularmente útil quando selecionamos a fonte de disparo como sendo o *software* (*bit* ADC12CTL0:ADC12SC). Neste caso, o programa pode ativar o *bit* ADC12SC pelo tempo necessário à amostragem. A conversão terá início somente quando esse *bit* for desativado pelo programa do usuário. Essa facilidade permite obter tempos de amostragem maiores que os oferecidos pelo *hardware* do conversor.

Com SHP = 1 temos o modo de amostragem temporizada. O tempo de amostragem é determinado por um *timer* interno, que é configurado de acordo com os *bits* SHT1x e SHT0x (registrador ADC12CTL0). Os *bits* SHT1x determinam o tempo de amostragem (em ciclos de *clock* do conversor) para as memórias ADC12MEM8 até ADC12MEM15. Os *bits* SHT0x determinam o tempo de amostragem para as memórias ADC12MEM0 até ADC12MEM7.

Nesse modo, quando selecionada a fonte de disparo como sendo o *software* e setado o *bit* ADC12CTL0:ADC12SC, ele é automaticamente apagado pelo *hardware* após iniciado o ciclo de amostragem.

O período mínimo de amostragem do sinal de entrada é determinado pela resistência interna da fonte do sinal e pode ser determinado pela seguinte equação:

$$t_{amostragem} = (R_S + 2000) * 9{,}011 * 40 * 10^{-12} + 800 * 10^{-9}$$

sendo:

- $t_{amostragem}$ - tempo mínimo de amostragem em segundos
- R_S - resistência interna da fonte de sinal em Ohms

Após a fase de amostragem, tem início a conversão do sinal pelo circuito SAR. São necessários 13 ciclos de *clock* (ADC12CLK) para que ela seja concluída.

A fonte de *clock* do conversor pode ser selecionada pelos *bits* ADC12SSELx (registrador ADC12CTL1), conforme a tabela:

ADC12SSELx	Fonte de *clock*
00	Oscilador interno do ADC12
01	ACLK
10	MCLK
11	SMCLK

Tabela 5-36

O oscilador interno do ADC12 opera a aproximadamente 5MHz (a margem média de tolerância é de 3,6 a 6,3 MHz).

A freqüência de *clock* selecionada pode ser dividida por um fator entre 1 e 8, conforme a seleção dos *bits* ADC12DIVx (registrador ADC12CTL1).

Muito cuidado deve ser tomado para que o conversor opere com uma freqüência de *clock* entre 450KHz e 6,3MHz. A operação fora desta faixa não é garantida pelo fabricante e pode estar sujeita a erros e perda de linearidade.

Durante um ciclo de amostragem/conversão, o *flag* ADC12BUSY (registrador ADC12CTL1) permanece setado, sendo apagado automaticamente após o término da conversão.

Completada a conversão, o resultado é armazenado em uma das 16 memórias do conversor (ADC12MEM0 a ADC12MEM15). A seleção da memória que receberá o resultado depende da forma como estão configurados os *bits* CONSEQx (registrador ADC12CTL1).

As memórias de conversão atuam praticamente como se fossem conversores independentes, permitindo selecionar o canal de entrada e as referências utilizadas na conversão. Também é possível a criação de seqüências predefinidas de conversão, utilizando as funcionalidades disponíveis nestas memórias.

Cada memória possui um registrador de controle (ADC12MTCLx) e um registrador de resultado (ADC12MEMx). O registrador de controle possui um *bit* EOS utilizado para sinalizar que aquela é a última memória de uma seqüência. Além disso, cada registrador de controle permite definir as referências (*bits* SREFx) e o canal de entrada (*bits* INCHx) que será utilizado naquela conversão.

Nos modelos MSP430FG43x existem doze canais analógicos de entrada, nos demais modelos, somente oito. A tabela 5-37 demonstra a seleção dos canais:

INCHx	Canal
0000	A0
0001	A1
0010	A2
0011	A3
0100	A4
0101	A5
0110	A6
0111	A7
1000	Ve_{REF+}
1001	V_{REF-} / Ve_{REF-}
1010	Diodo de temperatura
1011	$(V_{CC}-V_{SS})/2$
1100	A12 (modelos FG43x) $(V_{CC}-V_{SS})/2$ (demais modelos)
1101	A13 (modelos FG43x) $(V_{CC}-V_{SS})/2$ (demais modelos)
1110	A14 (modelos FG43x) $(V_{CC}-V_{SS})/2$ (demais modelos)
1111	A15 (modelos FG43x) $(V_{CC}-V_{SS})/2$ (demais modelos)

Tabela 5-37

A tabela 5-38 demonstra os modos possíveis para a configuração do conversor.

Observe que, quando opera em um modo de conversão seqüencial e no caso de nenhuma das memórias possuir um *bit* EOS setado, a seqüência pula da memória ADC12MEM15 para a memória ADC12MEM0, continuando a partir dela.

CONSEQx	MSC	Modo	Descrição
00	x	Um canal sem repetição	Os *bits* ADC12CTL1:CSTARTADDx indicam a memória de controle que receberá o resultado da conversão. O canal de entrada e a seleção da referência são feitos pelo registrador de controle daquela memória (ADC12MCTLx). O resultado é escrito na memória ADC12MEMx. Cada nova conversão necessita de um novo disparo. No caso de a fonte de disparo ser um dos canais dos *timers*, também é necessário que o *bit* ENC seja apagado e novamente setado antes da próxima conversão. O estado do *bit* MSC nesse modo é indiferente.
01	0	Seqüência de canais sem repetição	Uma seqüência de canais é convertida uma única vez. O início da seqüência é determinado pelos *bits* ADC12CTL1:CSTARTADDx, que apontam a memória ADC12MEMx inicial. A seqüência termina na memória cujo registrador de controle possui o *bit* EOS setado. Cada resultado é armazenado no respectivo registrador ADC12MEMx. Uma nova conversão da seqüência necessita de um novo disparo. No caso de a fonte de disparo ser um dos canais dos *timers*, também é necessário que o *bit* ENC seja apagado e novamente setado antes da próxima conversão.
10	1	Um canal repetidamente	Os *bits* ADC12CTL1:CSTARTADDx indicam a memória de controle que receberá o resultado da conversão. O canal de entrada e a seleção da referência são feitos pelo registrador de controle daquela memória (ADC12MCTLx). O resultado é escrito na memória ADC12MEMx. O canal é amostrado e convertido continuamente. Para encerrar as conversões, basta apagar o *bit* ENC.
11	1	Seqüência de canais repetidamente	Uma seqüência de canais é convertida repetidamente. O início da seqüência é determinado pelos *bits* ADC12CTL1:CSTARTADDx, que apontam a memória ADC12MEMx inicial. A seqüência termina na memória cujo registrador de controle possui o *bit* EOS setado. Cada resultado é armazenado no respectivo registrador ADC12MEMx. Para encerrar as conversões, basta apagar o *bit* ENC.

Tabela 5-38

5.15.1. Referências de Tensão

Cada uma das dezesseis memórias do ADC12 pode operar utilizando tanto uma das suas referências internas quanto uma ou duas referências externas.

As entradas de referência V_{R+} e V_{R-} permitem definir os limites superior e inferior de cada conversão, ou seja, o maior e o menor valor de tensão que o sistema pode converter.

Isso significa que, quando a tensão de entrada é igual ou maior que V_{R+}, a saída do conversor é igual a 4095 (0x0FFF) e quando a tensão de entrada é menor ou igual a V_{R-}, a saída do conversor é igual a 0.

Podemos selecionar a referência a ser utilizada pelos *bits* SREFx de cada um dos registradores de controle das memórias (ADC12MCTLx), conforme a tabela 5-39.

SREFx	V_{R+}	V_{R-}
000	AV_{CC}	AV_{SS}
001	V_{REF+}	AV_{SS}
010	Ve_{REF+}	AV_{SS}
011	Ve_{REF+}	AV_{SS}
100	AV_{CC}	V_{REF-}/Ve_{REF-}
101	V_{REF+}	V_{REF-}/Ve_{REF-}
110	Ve_{REF+}	V_{REF-}/Ve_{REF-}
111	Ve_{REF+}	V_{REF-}/Ve_{REF-}

Tabela 5-39

A opção V_{REF+} é proveniente do gerador de tensão de referência interno e pode ser ativada e desativada pelo *bit* REFON (registrador ADC12CTL0). A tensão de referência ativada pode ser selecionada entre dois valores: 1,5 ou 2,5 Volts. A seleção é feita pelo *bit* REF25V (registrador ADC12CTL0).

A tensão de referência interna está disponível externamente pelo pino V_{REF+} do microcontrolador. O fabricante recomenda a conexão desse pino a um capacitor de 10μF em paralelo com outro de 100nF, de forma a estabilizar a referência de tensão interna.

Além da tensão de referência interna, podemos utilizar uma tensão de referência externa aplicada ao pino Ve_{REF+} do *chip*, além de outra para a margem inferior, aplicada ao pino V_{REF-}/Ve_{REF-}. Alternativamente, também é possível utilizar como referência a própria tensão de alimentação do chip (AV_{CC} e AV_{SS}).

O resultado final da conversão pode ser expresso pela seguinte fórmula:

$$\text{Resultado}_{ADC} = 4095 * \frac{V_{in} - V_{R-}}{V_{R+} - V_{R-}}$$

5.15.2. Configuração

A configuração do ADC12 deve, de maneira geral, seguir estes passos:

1. O *bit* ADC12CTL0:ENC deve ser apagado.
2. Selecionam-se a fonte de *clock*, seu fator de divisão, modo e tempo de amostragem, referência de tensão, modo de conversão e fonte de disparo da conversão (registradores ADC12CTL0 e ADC12CTL1).
3. Selecionam-se a referência de tensão e o canal de entrada de cada memória utilizada (registradores ADC12MCTLx).
4. Habilita-se o ADC12 (ADC12CTL0:ENC = 1).
5. Configuram-se os pinos da porta, utilizados com o conversor, para a função alternativa (registrador PXSEL).

5.15.3. Sensor de Temperatura

Os microcontroladores MSP430 dotados de conversor A/D de 12 *bits* incluem um sensor de temperatura integrado à pastilha do *chip*.

A utilização do sensor é muito simples. Basta selecionar o canal de entrada número 10. Com ele selecionado, o sensor de temperatura é automaticamente ativado assim como a referência interna de temperatura.

O restante da configuração do ADC12 deve ser feito normalmente, como no caso da utilização de uma entrada externa.

A função típica de transferência do sensor pode ser expressa pela seguinte equação:

$$\text{TEMP}(^{\circ}\text{C}) = \frac{V_{TEMP} - 0{,}986}{0{,}00355}$$

Repare que para correta utilização do sensor de temperatura é importante utilizar um tempo de amostragem superior a 30μs.

Além disso, os valores medidos podem apresentar um *offset* que varia conforme o dispositivo, podendo ser necessária uma etapa de calibração antes de utilizar o sensor.

5.15.4. Interrupções

O ADC12 dispõe de 18 fontes de interrupção :

- Conversão completada em uma memória (o que resulta em 16 interrupções possíveis);
- Sobrescrita de um resultado;
- Estouro de tempo de conversão.

A interrupção de conversão completada é ativada sempre que a memória de conversão recebe um novo resultado do ADC. Neste caso, o *flag* ADC12IFGx é setado.

As interrupções das memórias de conversão podem ser individualmente habilitadas pelo registrador ADC12IE. Uma vez habilitada a interrupção da memória e estando o GIE em "1", a ativação do *flag* ADC12IFG correspondente àquela posição de memória fará com que o fluxo do programa seja desviado para o endereço apontado pelo vetor de interrupção do ADC12 (veja a tabela 5-1).

Outra fonte de interrupção do ADC12 é o indicador de erro de sobrescrita de um resultado na memória do conversor. Essa condição ocorre quando um novo valor é escrito em uma das memórias de conversão (ADC12MEMx) antes que o seu valor prévio seja lido. Essa interrupção não possui um *flag* acessível ao usuário e somente pode ser testada pelo registrador do gerador de vetor de interrupção (ADC12IV), conforme veremos a seguir. A sua habilitação pode ser feita pelo *bit* ADC12OVIE (registrador ADC12CTL0).

A última fonte de interrupção do ADC12 é a interrupção de estouro do tempo de conversão. Essa condição ocorre quando um sinal de disparo de conversão é detectado enquanto o conversor ainda está ocupado com uma conversão prévia (*bit* ADC12CTL1:ADC12BUSY = 1). A interrupção também não possui um *flag* acessível ao usuário e somente pode ser testada pelo registrador do gerador de vetor de interrupção (ADC12IV). A habilitação dessa interrupção é controlada pelo *bit* ADC12TOVIE (registrador ADC12CTL0).

Para facilitar e acelerar o tratamento das interrupções em uma RTI, o ADC12 inclui um gerador de vetores de interrupção. Para isso um código correspondente à fonte geradora do evento de interrupção é armazenado no registrador ADC12IV após o evento de interrupção (desde que ela esteja individualmente habilitada).

Dentro de uma RTI, a simples leitura do registrador ADC12IV faz com que o *flag* da interrupção causadora do desvio seja apagado. Caso mais de uma interrupção do módulo ocorra simultaneamente, a que tiver maior prioridade será tratada primeiro. Neste caso, somente o *flag* daquela interrupção é automaticamente apagado pelo *hardware*. As demais interrupções continuarão pendentes e serão tratadas na seqüência, uma após a outra.

5.15.5. Conexões do ADC12

O ADC12 pode utilizar os pinos apresentados na tabela 5-40.

O sinal de *clock* do conversor (ADC12CLK) pode ser disponibilizado externamente. Para isso é necessário configurar o pino como saída (P2DIR:x = 1) e para a função alternativa (P2SEL:x = 1).

Sinal	Pino
A0	P6.0
A1	P6.1
A2	P6.2
A3	P6.3
A4	P6.4
A5	P6.5
A6	P6.6
A7	P6.7
A12	P5.1 (somente nos modelos FG43x)
A13	P5.0 (somente nos modelos FG43x)
A14	P4.7 (somente nos modelos FG43x)
A15	P4.6 (somente nos modelos FG43x)
ADC12CLK	P2.6 (família 1xx) P2.7 (família 4xx)
V_{REF+}	V_{REF+}
Ve_{REF+}	Ve_{REF+}
V_{REF-} / Ve_{REF-}	V_{REF-} / Ve_{REF-}

Tabela 5-40

5.15.6. Registradores

O ADC12 utiliza os seguintes registradores para a sua configuração e operação:

- **ADC12CTL0** e **ADC12CTL1** - configuração do ADC12;
- **ADC12MEM0** a **ADC12MEM15** - resultado da conversão;
- **ADC12MCTL0** a **ADC12MCTL15** - configuração das memórias de conversão;
- **ADC12IFG** - *flags* de interrupção das memórias de conversão;
- **ADC12IE** - habilitação da interrupção das memórias de conversão;
- **ADC12IV** - gerador de vetores de interrupção do ADC12;
- **PxSEL** - configuração dos pinos de entrada do conversor para o modo analógico.

5.15.6.1. ADC12CTL0

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01A0	ADC12CTL0	Leitura / Escrita	SHT1x*				SHT0x*			
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura / Escrita	MSC*	REF2_5V*	REFON*	ADC12ON*	ADC12 OVIE	ADC12 TOVIE	ENC	ADC12SC
		Reset	0	0	0	0	0	0	0	0

* Esses *bits* somente podem ser alterados com ENC = 0.

SHT1x e SHT0x - seleção do tempo de amostragem dos canais 0 a 7 (SHT0) e dos canais 8 a 15 (SHT1):

SHTx	Tempo de Amostragem (ADC12CLK)	Símbolo
0000	4	**SHT0_0 / SHT1_0**
0001	8	**SHT0_1 / SHT1_1**
0010	16	**SHT0_2 / SHT1_2**
0011	32	**SHT0_3 / SHT1_3**
0100	64	**SHT0_4 / SHT1_4**
0101	96	**SHT0_5 / SHT1_5**
0110	128	**SHT0_6 / SHT1_6**
0111	192	**SHT0_7 / SHT1_7**
1000	256	**SHT0_8 / SHT1_8**
1001	384	**SHT0_9 / SHT1_9**
1010	512	**SHT0_10 / SHT1_10**
1011	768	**SHT0_11 / SHT1_11**
1100	1024	**SHT0_12 / SHT1_12**
1101	1024	**SHT0_13 / SHT1_13**
1110	1024	**SHT0_14 / SHT1_14**
1111	1024	**SHT0_15 / SHT1_15**

MSC - seleção de múltiplas conversões (somente nos modos de seqüência ou repetitivos):
0 - modo de conversão única (um evento de disparo de conversão provoca somente uma conversão);
1 - modo de conversão repetitiva (um evento de disparo provoca o início de múltiplas conversões) (símbolo **MSC**).

REF2_5V - seleção da tensão de saída do gerador de tensão de referência (somente quando REFON = 1, ou seja, quando o gerador de referência estiver ligado):
0 - 1,5 Volts;
1 - 2,5 Volts (símbolo **REF2_5V**).

REFON - ativação do gerador de tensão de referência interno:
0 - desligado;
1 - ligado (símbolo **REFON**).

ADC12ON - controle liga/desliga do módulo ADC12:
0 - desligado;
1 - ligado (símbolo **ADC12ON**).

ADC12OVIE - controle de habilitação da interrupção de sobrescrita do resultado da conversão:
0 - desabilitada;
1 - habilitada (símbolo **ADC12OVIE**).

ADC12TOVIE - controle de habilitação da interrupção de estouro de tempo de conversão (uma nova amostragem/conversão foi disparada antes que a anterior fosse completada):
0 - nenhuma interrupção;
1 - habilitada (símbolo **ADC12TOVIE**).

ENC - habilitação da conversão:
0 - conversor desabilitado;
1 - conversor habilitado (símbolo **ENC**).

ADC12SC - início da conversão. Esse *bit*, quando setado, provoca o início de uma conversão (desde que a fonte de disparo selecionada seja o *software*). O *bit* é automaticamente apagado pelo *hardware* após o início da amostragem:
0 - nenhuma conversão;
1 - iniciar amostragem/conversão (símbolo **ADC12SC**).

5.15.6.2. ADC12CTL1

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01A2	ADC12CTL1	Leitura	CSTARTADDx				SHSx		SHP*	ISSH*
		Escrita								
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	ADC12DIVx*			ADC12SSELx*		CONSEQx*		ADC12
		Escrita								BUSY
		Reset	0	0	0	0	0	0	0	0

* Esses bits somente podem ser alterados com ENC = 0.

CSTARTADDx - seleção do endereço do registrador para o armazenamento dos resultados da conversão:

CSTARTADDx	Registrador inicial	Símbolo
0000	ADC12MEM0	**CSTARTADD_0**
0001	ADC12MEM1	**CSTARTADD_1**
0010	ADC12MEM2	**CSTARTADD_2**
0011	ADC12MEM3	**CSTARTADD_3**
0100	ADC12MEM4	**CSTARTADD_4**
0101	ADC12MEM5	**CSTARTADD_5**
0110	ADC12MEM6	**CSTARTADD_6**
0111	ADC12MEM7	**CSTARTADD_7**
1000	ADC12MEM8	**CSTARTADD_8**
1001	ADC12MEM9	**CSTARTADD_9**
1010	ADC12MEM10	**CSTARTADD_10**
1011	ADC12MEM11	**CSTARTADD_11**
1100	ADC12MEM12	**CSTARTADD_12**
1101	ADC12MEM13	**CSTARTADD_13**
1110	ADC12MEM14	**CSTARTADD_14**
1111	ADC12MEM15	**CSTARTADD_15**

SHSx - seleção do disparo do conversor:

SHSx	Fonte de disparo	Símbolo
00	*software* (*bit* ADC12CTL0:ADC12SC)	**SHS_0**
01	*timer* A (saída TA1)	**SHS_1**
10	*timer* B (saída TB0)	**SHS_2**
11	*timer* B (saída TB1)	**SHS_3**

SHP - seleção do modo de amostragem:

0 - modo pulsado (a amostragem dura o tempo que o sinal de disparo da conversão permanecer ativo);

1 - modo temporizado (a amostragem dura o tempo definido pelos *bits* ADC12CTL0:SHTxx) (símbolo **SHP**).

ISSH - inversão do sinal início de amostragem e conversão:

0 - o sinal de disparo não é invertido;

1 - o sinal de disparo é invertido (símbolo **ISSH**).

ADC12DIV - seleção do divisor do *clock* do ADC12:

ADC12DIV	Fator de divisão	Símbolo
000	1	**ADC12DIV_0**
001	2	**ADC12DIV_1**
010	3	**ADC12DIV_2**
011	4	**ADC12DIV_3**
100	5	**ADC12DIV_4**
101	6	**ADC12DIV_5**
110	7	**ADC12DIV_6**
111	8	**ADC12DIV_7**

ADC12SSELx - seleção da fonte de *clock* do ADC12:

ADC12SSELx	Fonte de *clock*	Símbolo
00	ADC12OSC	**ADC12SSEL_0**
01	ACLK	**ADC12SSEL_1**
10	MCLK	**ADC12SSEL_2**
11	SMCLK	**ADC12SSEL_3**

CONSEQx - inversão do sinal início de amostragem e conversão:

CONSEQx	Modo de conversão	Símbolo
00	Um canal, conversão simples	**CONSEQ_0**
01	Seqüência de canais	**CONSEQ_1**
10	Um canal, repetitivo	**CONSEQ_2**
11	Seqüência de canais, repetitivo	**CONSEQ_3**

ADC12BUSY - indicador de atividade do ADC12 (somente leitura):

0 - nenhuma operação em andamento;

1 - uma operação de amostragem ou conversão está em andamento (símbolo **ADC12BUSY**).

5.15.6.3. ADC12MCTLx

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0080	ADC12MCTL0	Leitura	EOS*	SREFx*			INCHx*			
0x0081	ADC12MCTL1									
0x0082	ADC12MCTL2									
0x0083	ADC12MCTL3									
0x0084	ADC12MCTL4									
0x0085	ADC12MCTL5	Escrita								
0x0086	ADC12MCTL6									
0x0087	ADC12MCTL7									
0x0088	ADC12MCTL8									
0x0089	ADC12MCTL9									
0x008A	ADC12MCTL10									
0x008B	ADC12MCTL11									
0x008C	ADC12MCTL12									
0x008D	ADC12MCTL13	*Reset*	0	0	0	0	0	0	0	0
0x008E	ADC12MCTL14									
0x008F	ADC12MCTL15									

* Esses bits somente podem ser alterados com ENC = 0.

EOS - seleção de fim da seqüência:

0 - esse canal não é o último da seqüência;

1 - esse canal é o último da seqüência (símbolo **EOS**).

SREFx - seleção das referências do conversor:

SREFx	V_{R+}	V_{R-}	Símbolo
000	AV_{CC}	AV_{SS}	**SREF_0**
001	V_{REF+}	AV_{SS}	**SREF_1**
010	Ve_{REF+}	AV_{SS}	**SREF_2**
011	Ve_{REF+}	AV_{SS}	**SREF_3**
100	AV_{CC}	V_{REF-}/Ve_{REF-}	**SREF_4**
101	V_{REF+}	V_{REF-}/Ve_{REF-}	**SREF_5**
110	Ve_{REF+}	V_{REF-}/Ve_{REF-}	**SREF_6**
111	Ve_{REF+}	V_{REF-}/Ve_{REF-}	**SREF_7**

INCHx - seleção do canal de entrada:

INCHx	Canal	Símbolo
0000	A0	**INCH_0**
0001	A1	**INCH_1**
0010	A2	**INCH_2**
0011	A3	**INCH_3**
0100	A4	**INCH_4**
0101	A5	**INCH_5**
0110	A6	**INCH_6**
0111	A7	**INCH_7**
1000	Ve_{REF+}	**INCH_8**
1001	V_{REF-} / Ve_{REF-}	**INCH_9**
1010	Diodo de temperatura	**INCH_10**
1011	$(V_{CC}-V_{SS})/2$	**INCH_11**
1100	A12 (modelos FG43x) $(V_{CC}-V_{SS})/2$ (demais modelos)	**INCH_12**
1101	A13 (modelos FG43x) $(V_{CC}-V_{SS})/2$ (demais modelos)	**INCH_13**
1110	A14 (modelos FG43x) $(V_{CC}-V_{SS})/2$ (demais modelos)	**INCH_14**
1111	A15 (modelos FG43x) $(V_{CC}-V_{SS})/2$ (demais modelos)	**INCH_15**

5.15.6.4. ADC12MEMx

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0140 0x0142 0x0144 0x0146 0x0148 0x014A 0x014C 0x014E 0x0150 0x0152 0x0154 0x0156 0x0158 0x015A 0x015C 0x015E	ADC12MEM0 ADC12MEM1 ADC12MEM2 ADC12MEM3 ADC12MEM4 ADC12MEM5 ADC12MEM6 ADC12MEM7 ADC12MEM8 ADC12MEM9 ADC12MEM10 ADC12MEM11 ADC12MEM12 ADC12MEM13 ADC12MEM14 ADC12MEM15	Leitura Escrita	0	0	0	0	Resultado (4 *bits* MSB)			
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura Escrita	Resultado (8 *bits* LSB)							
		Reset	0	0	0	0	0	0	0	0

5.15.6.5. ADC12IE

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01A6	ADC12IE	Leitura	ADC12IE	ADC12IE	ADC12IE	ADC12IE	ADC12IE	ADC12IE	ADC12IE9	ADC12IE8
		Escrita	15	14	13	12	11	10		
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	ADC12IE7	ADC12IE6	ADC12IE5	ADC12IE4	ADC12IE3	ADC12IE2	ADC12IE1	ADC12IE0
		Escrita								
		Reset	0	0	0	0	0	0	0	0

5.15.6.6. ADC12IFG

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01A4	ADC12IFG	Leitura	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG
		Escrita	15	14	13	12	11	10	9	8
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG	ADC12IFG
		Escrita	7	6	5	4	3	2	1	0
		Reset	0	0	0	0	0	0	0	0

5.15.6.7. ADC12IV

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01A8	ADC12IV	Leitura	0	0	0	0	0	0	0	0
		Escrita	-	-	-	-	-	-	-	-
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	0	0	ADC12IVx					0
		Escrita	-	-	-					-
		Reset	0	0	0	0	0	0	0	0

ADC12IV	Fonte de interrupção	*Flag* de interrupção	Prioridade
0x00	Nenhuma interrupção	-	
0x02	Sobrescrita de resultado	-	Maior
0x04	Estouro do tempo de conversão	-	
0x06	conversão na memória 0	ADC12IFG:ADC12IFG0	
0x08	conversão na memória 1	ADC12IFG:ADC12IFG1	
0x0A	conversão na memória 2	ADC12IFG:ADC12IFG2	
0x0C	conversão na memória 3	ADC12IFG:ADC12IFG3	
0x0E	conversão na memória 4	ADC12IFG:ADC12IFG4	
0x10	conversão na memória 5	ADC12IFG:ADC12IFG5	
0x12	conversão na memória 6	ADC12IFG:ADC12IFG6	
0x14	conversão na memória 7	ADC12IFG:ADC12IFG7	
0x16	conversão na memória 8	ADC12IFG:ADC12IFG8	
0x18	conversão na memória 9	ADC12IFG:ADC12IFG9	

ADC12IV	Fonte de interrupção	*Flag* de interrupção	Prioridade
0x1A	conversão na memória 10	ADC12IFG:ADC12IFG10	
0x1C	conversão na memória 11	ADC12IFG:ADC12IFG11	
0x1E	conversão na memória 12	ADC12IFG:ADC12IFG12	
0x20	conversão na memória 13	ADC12IFG:ADC12IFG13	
0x22	conversão na memória 14	ADC12IFG:ADC12IFG14	
0x24	conversão na memória 15	ADC12IFG:ADC12IFG15	Menor

Tabela 5-41

5.15.7. Exemplo de Utilização

O exemplo seguinte demonstra a utilização do ADC12. Dois canais de entrada: A0 e A1 são amostrados e convertidos repetidamente. O programa lê os resultados e caso a tensão em A0 seja maior que a tensão em A1, um LED conectado ao pino P1.0 é aceso; caso contrário, o LED é apagado. O programa foi testado em um *chip* MSP430F449, utilizando a placa Microlab X1. Os *trimpots* presentes na placa foram conectados aos pinos P6.0 e P6.1.

Para utilização com um *chip* da família 1xx, basta alterar o arquivo de inclusão no início do programa e o dispositivo na tela de opções do projeto. Nenhuma outra alteração será necessária.

```
#include <io430x44x.h>

void main(void)
{
   WDTCTL = WDTPW + WDTHOLD;     // desliga o watchdog
   // Configura o ADC12:
   // modo sequencia repetitiva de canais
   // amostragem temporizada
   ADC12CTL1 = CONSEQ_3 + SHP;
   // configura a memória 0 para leitura do canal A0
   ADC12MCTL0 = 0;
   // configura a memória 1 para leitura do canal A1
   // e como a última memória da seqüência (EOS=1)
   ADC12MCTL1 = INCH_1 + EOS;
   // ativa o ADC12, configura múltiplas conversões
   // habilita e inicia as conversões
   ADC12CTL0 = MSC + ADC12ON + ENC + ADC12SC;
   // configura os pinos P6.0 e P6.1 para a função alternativa
   // (ADC12)
   P6SEL = 0x03;
   // configura os pinos da porta 1 como saída
   P1DIR = 0xff;
   // coloca todos os pinos da porta 1 em nível "0"
   P1OUT = 0;
   while (1)
   {
      // se o valor lido da memória 0 for maior que
      // o valor lido da memória 1
      if (ADC12MEM0 > ADC12MEM1)
         P1OUT_bit.P1OUT_0 = 1;  // acende o led
      else                       // senão
         P1OUT_bit.P1OUT_0 = 0;  // apaga o led
   }
}
```

Exemplo 5-13

5.16. Conversor Digital-Analógico

O conversor digital/analógico de 12 *bits* (DAC12) é um periférico encontrado nos modelos MSP430F15x, 16x, 42x0 e FG43x, capaz de converter uma informação binária em uma tensão de saída proporcional.

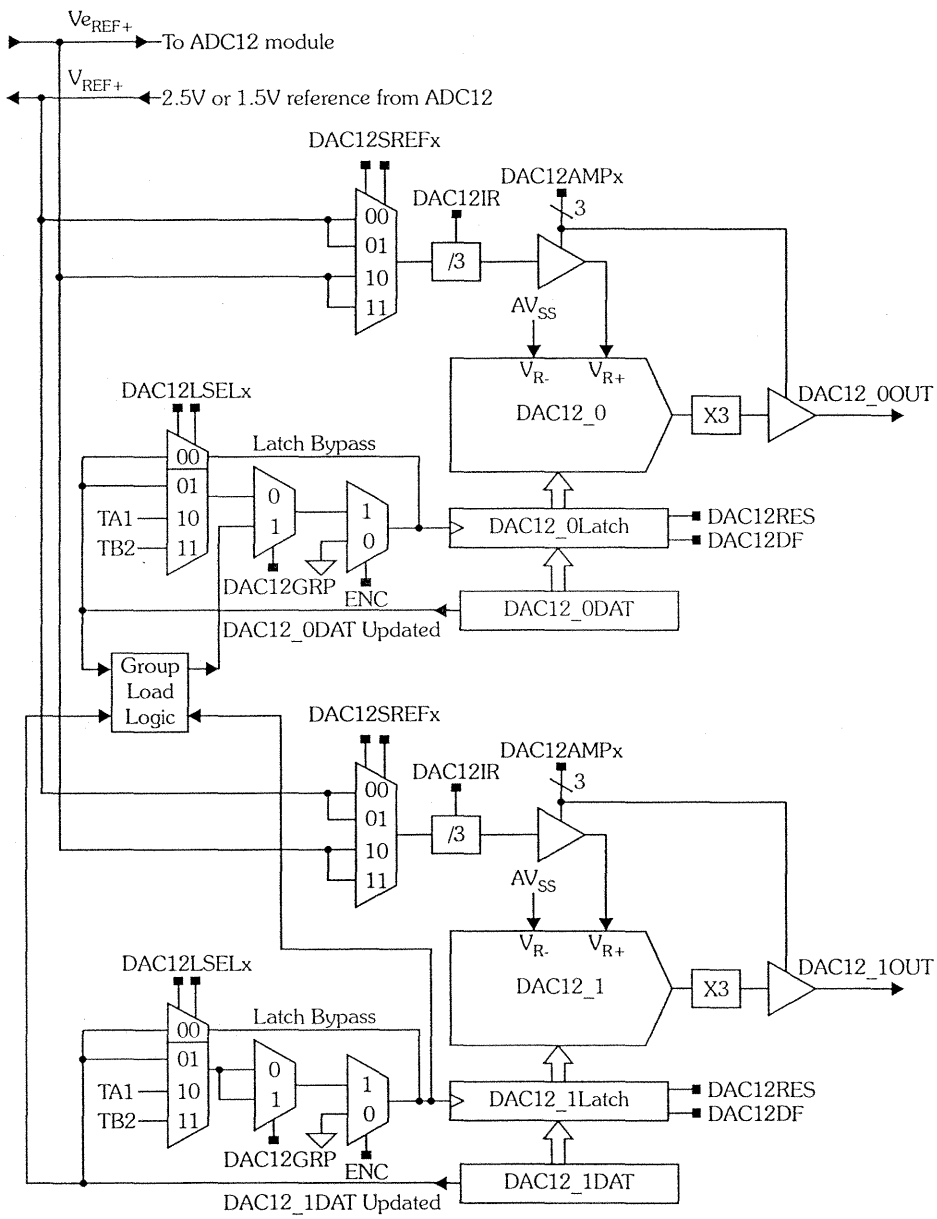


Figura 5-46

As principais características do DAC12 são:

- Saída de 8 ou 12 *bits* monotônica;
- Velocidade programável (5, 2 ou 1μs), permitindo obter a melhor relação entre consumo de corrente e velocidade de conversão;
- Baixo consumo de corrente (entre 50 e 700μA, dependendo da velocidade de operação);
- Referência interna de 15 ou 2,5 Volts (proveniente do ADC12) ou externa;
- Autocalibração para correção da tensão de *offset* de saída;
- Um canal (nos modelos 42x0) ou dois canais (nos modelos 15x, 16x e FG43x);
- Possibilidade de agrupar os dois canais para atualização das saídas simultaneamente.

O funcionamento do DAC12 é bastante simples: o coração do circuito é um conversor D/A que pode operar em 8 ou 12 *bits*. O valor digital a ser convertido deve ser escrito no registrador DAC12_xDAT.

Os *bits* DAC12LSELx (registrador DAC12_xCTL) selecionam o modo de disparo da conversão:

00 - a conversão é iniciada pela escrita no registrador DAC12_xDAT. O estado do *bit* DAC12ENC (registrador DAC12_xCTL) é ignorado.

01 - a conversão é iniciada pela escrita no registrador DAC12_xDAT. Nesse modo, quando se utilizam canais DAC agrupados, somente após todos os registradores DAC12_xDAT terem sido escritos é que a saída deles é atualizada.

10 - a conversão é iniciada por uma borda de subida do sinal de saída do canal 1 do *timer* A (TA1).

11 - a conversão é iniciada por uma borda de subida do sinal de saída do canal 2 do *timer* B (TB2).

Nos três últimos casos, é necessário que o *bit* DAC12ENC (registrador DAC12_xCTL) esteja setado para que o valor seja carregado no conversor.

O conversor necessita de um período de tempo (*settling time*) para que a sua tensão de saída seja atualizada pelo valor carregado na sua entrada. Esse período de tempo depende da configuração dos *buffers* de referência e de saída: quanto maior a corrente fornecida por eles, mais rápida será a conversão (menor o *settling time*).

A configuração desses *buffers* é selecionada pelos *bits* DAC12AMPx, conforme a tabela:

DAC12AMPx	*Buffer* da Referência	*Buffer* de Saída
000	desligado	DAC desligado, saída em alta impedância
001	desligado	DAC desligado, saída em 0V
010	baixa velocidade/corrente	baixa velocidade/corrente
011	baixa velocidade/corrente	média velocidade/corrente
100	baixa velocidade/corrente	alta velocidade/corrente
101	média velocidade/corrente	média velocidade/corrente
110	média velocidade/corrente	alta velocidade/corrente
111	alta velocidade/corrente	alta velocidade/corrente

Tabela 5-42

Quando os *bits* DAC12AMPx estão em "0", o módulo está desativado e os seus pinos de saída podem ser utilizados para outras funções.

Qualquer outra configuração desses *bits* ativa o módulo que assume o controle dos pinos de saída, configurando-os para a função DAC, independentemente da configuração dos registradores PxSEL e PxDIR que os controlam.

O sinal de saída pode ainda ser amplificado internamente, por um amplificador com ganho 3. Basta apagar o *bit* DAC12IR (registrador DAC12_xCTL). Quando ele está setado, o ganho do amplificador de saída é unitário.

O valor da tensão de saída pode então ser calculado pela seguinte equação:

$$V_{OUT} = Vref * MULT * \frac{DAC12_xDAT}{2^{RES}}$$

sendo:

- Vref - é o valor da tensão de referência do conversor;
- MULT - é o fator de multiplicação da tensão de saída (1 ou 3);
- DAC12_xDAT - é o valor carregado no conversor;
- RES - é o número de *bits* de resolução (8 ou 12).

A resolução do conversor, selecionada pelo *bit* DAC12RES (registrador DAC12_xCTL): com DAC12RES = 0, temos uma resolução de 12 *bits* e nesse caso, a tensão de saída atinge o seu nível máximo quando o valor carregado no conversor for igual a 4.095 (0x0FFF). Com DAC12RES = 1, temos uma resolução de 8 *bits*, o que implica que a tensão de saída alcança o seu valor máximo quando o conversor for carregado com o valor 0x00FF.

O formato de dado do conversor também influencia o valor da tensão de saída: o *bit* DAC12DF (registrador DAC12_xCTL) seleciona se o conversor vai trabalhar no modo binário puro ou utilizando o formato complemento de dois (valores sinalizados).

Uma facilidade adicional disponível nos modelos com mais de um canal DAC12 é a possibilidade de agrupar os canais de forma que todos atualizem simultaneamente. Para utilizar essa facilidade, basta setar o *bit* DAC12GRP no registrador DAC12_0CTL.

Com os canais agrupados, a sua saída somente será atualizada após a escrita nos registradores DAC12_0DAT e DAC12_1DAT. A ordem de escrita é indiferente. De qualquer modo, a saída dos DACs somente será atualizada quando ambos os registradores tiverem sido escritos e cada nova atualização necessita de nova escrita em ambos os registradores.

Quando agrupados, o modo de disparo de conversão de ambos os canais (DAC12_xCTL:DAC12LSELx) deve ser configurado para um valor maior que zero, pois quando DAC12LSELx = 0, a atualização dos DACs ocorre imediatamente após a escrita no registrador DAC12_xDAT. Além disso, é necessário que ambos os canais estejam também habilitados (DAC12ENC = 1).

Repare, no entanto, que mesmo que a fonte de disparo da conversão seja selecionada para uma das saídas dos *timers* (DAC12LSELx = 2 ou 3), ainda assim, as saídas dos DACs somente serão atualizadas após a escrita em ambos os registradores DAC12_xDAT. Neste caso, na próxima borda de subida do sinal de saída do *timer*, após a escrita em ambos os registradores DAC12_xDAT, as saídas dos DACs serão atualizadas.

5.16.1. Referências de Tensão

O DAC12 pode trabalhar com referências de tensão externas ou internas. A referência externa é a tensão aplicada ao pino Ve_{REF+}, ao passo que a referência interna é proveniente do gerador de referência do ADC12. A seleção da tensão de referência é feita pelos bits DAC12SREFx (registrador DAC12_xCTL).

Observe que, para utilizar a referência do ADC12, é necessário que ela esteja ativada e adequadamente configurada. Para maiores detalhes veja o tópico 5.15.1 na página 255.

5.16.2. Configuração

A configuração de cada canal do DAC12 deve seguir estes passos básicos:

1. O registrador DAC12_xCTL deve ser configurado (seleção da tensão de referência, fator de multiplicação da tensão de saída, ganho de corrente e velocidade dos *buffers* internos, interrupções, etc.). Lembre-se de que vários *bits* somente podem ser configurados com o *bit* DAC12ENC = 0.
2. Caso seja utilizado o gerador interno da tensão de referência, ele deve ser configurado no registrador ADC12CTL0.
3. Não é necessário alterar nenhuma configuração para os pinos de saída do DAC, já que ativado o DAC assume o controle desses pinos.

5.16.3. Operação com DMA

O DAC12 pode ser controlado pelo módulo DMA, quando estiver disponível. Neste caso, o controlador de DMA pode ser utilizado para ler um bloco de memória e enviar seqüencialmente esses dados para o DAC, permitindo gerar sinais gravados previamente na memória.

Uma precaução a ser tomada no que diz respeito à velocidade com que o DMA atualiza os dados do DAC: se a velocidade for maior que o tempo de conversão dele, a tensão de saída não será reproduzida adequadamente.

5.16.4. Interrupções

O módulo DAC dispõe de um *flag* de interrupção (DAC12_xCTL:DAC12IFG) que é setado sempre que um novo dado é carregado no *latch* do conversor, ou seja, quando uma conversão D/A tem início. Isso indica que o conversor está pronto para receber um novo dado a ser convertido.

O *flag* somente é setado quando a fonte de disparo da conversão (DAC12LSELx) está configurada com um valor maior que zero.

A interrupção do DAC pode ser individualmente habilitada pelo *bit* DAC12IE (registrador DAC12_xCTL).

5.16.5. Conexões do DAC12

O DAC12 pode utilizar os seguintes pinos:

Sinal	Pino
DAC0	P6.6 Ve_{REF+}(modelos FG43x, vide observação a seguir)
DAC1	P6.7 P5.1 (modelos FG43x, vide observação a seguir)
V_{REF+}	V_{REF+}
Ve_{REF+}	Ve_{REF+}

Tabela 5-43

Lembre-se de que, quando o DAC está ativado, os seus pinos de saída passam a ser automaticamente controlados por ele, independentemente da configuração dos registradores PxSEL e PxDIR.

Nos MSP430F43x, o *bit* DAC12_xCTL:DAC12OPS permite selecionar pinos de saída alternativos para os dois canais DAC.

5.16.6. Registradores do DAC12

Cada DAC12 utiliza dois registradores para a sua operação:

- **DAC12_0CTL** e **DAC12_1CTL** -controle e configuração dos canais;
- **DAC12_0DAT** e **DAC12_1DAT** -armazenam o dado a ser convertido.

5.16.6.1. DAC12_xCTL

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01C0 0x01C2	DAC12_0CTL DAC12_1CTL	Leitura Escrita	DAC12 OPS*	DAC12SREFx*		DAC12 RES*	DAC12LSELx*		DAC12 CALON*	DAC12IR*
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura Escrita	DAC12AMPx*			DAC12DF	DAC12IE	DAC12IFG	DAC12 ENC	DAC12 GRP
		Reset	0	0	0	0	0	0	0	0

* Esses *bits* somente podem ser alterados com DAC12ENC = 0.

DAC12OPS - seleção dos pinos de saída do DAC (esse *bit* somente está disponível nos modelos FG43x e deve ser mantido em zero nos demais *chips*):

0 - a saída do DAC0 é feita pelo pino P6.6 e do DAC1 pelo pino P6.7;

1 - a saída do DAC0 é feita pelo pino Ve_{REF+} e do DAC1 pelo pino P5.1 (símbolo **DAC12OPS**).

DAC12SREFx - seleção da referência de tensão do DAC:

00 - Referência interna V_{REF+} (símbolo **DAC12SREF_0**);

01 - Referência interna V_{REF+} (símbolo **DAC12SREF_1**);

10 - Referência externa Ve_{REF+} (símbolo **DAC12SREF_2**);

11 - Referência externa Ve_{REF+} (símbolo **DAC12SREF_3**).

DAC12RES - resolução do DAC:

0 - 12 *bits*;

1 - 8 *bits* (símbolo **DAC12RES**).

DAC12LSELx- seleção do evento de disparo da carga do *latch* do DAC (é necessário que o *bit* DAC12ENC esteja setado para que a saída do DAC seja atualizada, exceto quando o DAC12LSELx = 0):

00 - a carga é realizada no instante da escrita no registrador DAC12_xDAT (o estado do DAC12ENC é ignorado) (símbolo **DAC12LSEL_0**);

01 - a carga é realizada no instante da escrita no registrador DAC12_xDAT ou, no caso do agrupamento dos dois canais, quando todos os registradores DAC12_xDAT do grupo foram escritos (símbolo **DAC12LSEL_1**);

10 - a carga é realizada na borda de subida do sinal de saída do canal 1 do *timer* A (TA1) (símbolo **DAC12LSEL_2**);

11 - a carga é realizada na borda de subida do sinal de saída do canal 2 do *timer* B (TB2) (símbolo **DAC12LSEL_3**).

DAC12CALON - calibração do DAC. Esse *bit* provoca o início da seqüência de calibração de *offset* do DAC e é automaticamente apagado ao final desse processo:

0 - calibração inativa;

1 - iniciar calibração / calibração em curso (símbolo **DAC12CALON**).

DAC12IR - seleção da faixa de saída do DAC:

0 - o fundo de escala de saída é igual a três vezes a referência de tensão;

1 - o fundo de escala de saída é igual à referência de tensão (símbolo **DAC12IR**).

DAC12AMPx - seleção do comportamento dos *buffers* de entrada e de saída do DAC:

DAC12AMPx	*Buffer* da Referência	*Buffer* de Saída	*Símbolo*
000	desligado	DAC desligado, saída em alta impedância	**DAC12AMP_0**
001	desligado	DAC desligado, saída em 0V	**DAC12AMP_1**
010	baixa velocidade/corrente	baixa velocidade/corrente	**DAC12AMP_2**
011	baixa velocidade/corrente	média velocidade/corrente	**DAC12AMP_3**
100	baixa velocidade/corrente	alta velocidade/corrente	**DAC12AMP_4**
101	média velocidade/corrente	média velocidade/corrente	**DAC12AMP_5**
110	média velocidade/corrente	alta velocidade/corrente	**DAC12AMP_6**
111	alta velocidade/corrente	alta velocidade/corrente	**DAC12AMP_7**

DAC12DF - formato dos dados de entrada do DAC:

0 - binário puro;

1 - complemento de 2 (símbolo **DAC12DF**).

DAC12IE - habilitação da interrupção do DAC:

0 - desabilitada;

1 - habilitada (símbolo **DAC12IE**).

DAC12IFG - sinalizador de interrupção do DAC:

0 - nenhuma interrupção;

1 - interrupção pendente (símbolo **DAC12IFG**).

DAC12ENC - ativação da conversão do DAC. Se os *bits* DAC12LSELx forem diferentes de zero, o DAC só realiza uma conversão se DAC12ENC = 1. Se DAC12LSELx = 0, o DAC realiza conversões pela simples escrita no registrador DAC12_xDAT, neste caso, o DAC12ENC é ignorado:

0 - DAC12 desabilitado;

1 - DAC12 habilitado para conversões (símbolo **DAC12ENC**).

DAC12GRP - agrupamento dos DACs. Quando ativado, agrupa o DAC atual com o próximo DAC disponível (na seqüência). Somente aplicável ao DAC12_0:

0 - não agrupado;

1 - agrupado com o próximo DAC (símbolo **DAC12GRP**).

5.16.6.2. DAC12_xDAT

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x01C8 0x01CA	DAC12_0DAT DAC12_1DAT	Leitura Escrita	0	0	0	0	4 *bits* MSB do valor a ser convertido			
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura Escrita	8 *bits* LSB do valor a ser convertido							
		Reset	0	0	0	0	0	0	0	0

5.16.7. Exemplo de Configuração

No exemplo seguinte utilizamos um MSP430F169 para gerar uma tensão de aproximadamente 1 Volt na saída do DAC0.

Vamos utilizar a referência interna de 1,5 Volts do *chip* e a multiplicação "x3" do sinal de saída do DAC. Neste caso, o valor a ser carregado no DAC12_0DAT deve ser igual a:

$$DAC12_0DAT = \frac{2^{12} * V_{OUT}}{V_{REF} * MULT} = \frac{4096 * 1}{1{,}5 * 3} = 910$$

```
#include <io430x16x.h>

void main(void)
{
   ADC12CTL0 = REFON;                    // liga a referência do ADC12
   DAC12_0CTL = DAC12AMP_7 + DAC12ENC;   // configura o DAC12
   DAC12_0DAT = 910;                     // converte o valor
   while (1);
}
```

Exemplo 5-14

5.17. Controlador de LCD

Os MSP430 da família 4xx incluem um controlador de *displays* de cristal líquido (LCD) com as seguintes características:

- Controle de 96, 128 ou 160 segmentos, conforme o modelo do microcontrolador;
- Suporte a *displays* estáticos e multiplexados;
- Baixo consumo e operação em diversos modos de baixa potência (desde que o temporizador básico (*timer* 1) continue em funcionamento).

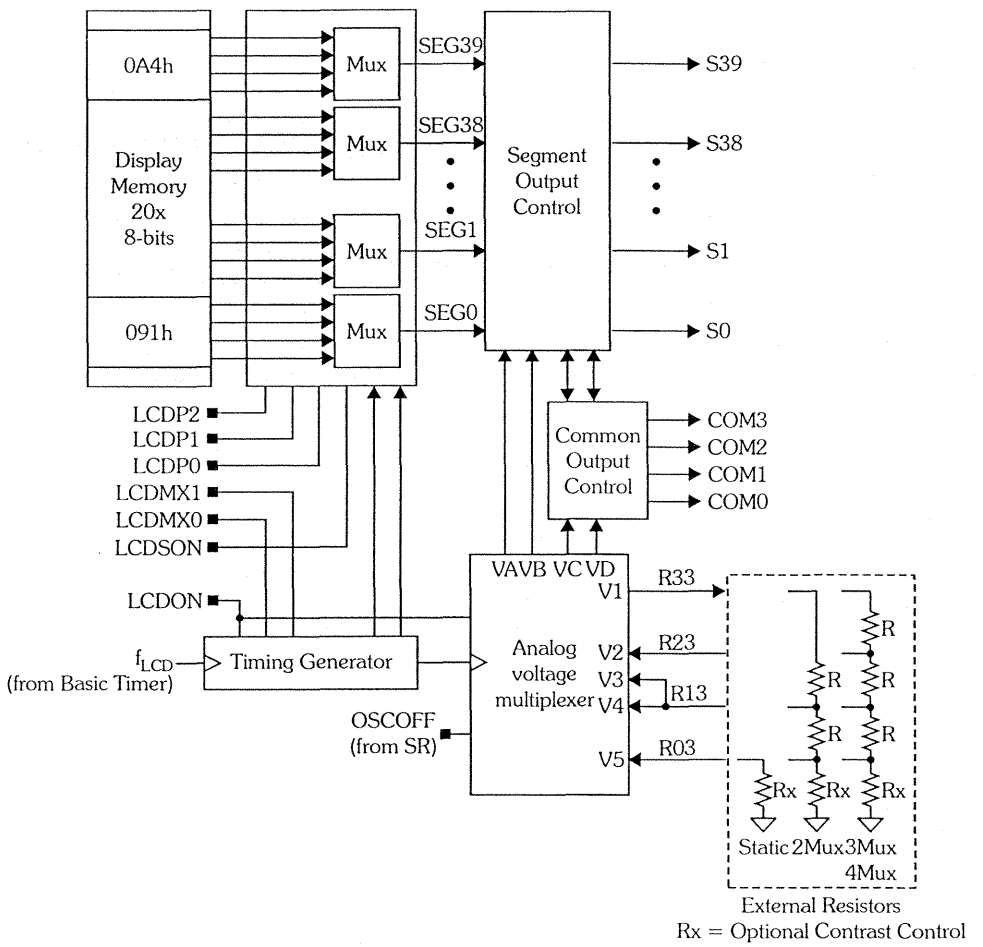


Figura 5-47

O funcionamento do controlador de LCD é muito simples: um circuito temporizador, cuja fonte de *clock* é o temporizador básico (tópico 5.7), gera os sinais de temporização necessários para a geração da tensão de excitação dos segmentos do LCD. O mesmo circuito gera também os sinais de controle para os multiplexadores digitais, quando se utilizam *displays* LCD multiplexados.

A tensão de excitação, aplicada nos chamados planos de fundo ou *back-planes*, é obtida por uma associação de resistores conectados a um multiplexador analógico. O multiplexador possui quatro pinos de conexão, denominados R03, R13, R23 e R33 (veja a figura 5-47).

O pino R33 é utilizado para alimentação da rede de resistores e possui potencial igual ao V_{CC} quando o controlador está ativo. Esse nível de tensão é chamado internamente de V1.

O pino R23 é conectado à malha de resistores de forma que a tensão sobre ele é uma fração da tensão V_{CC}, chamada internamente de V2.

O pino R13 também está conectado à malha de resistores, mas em um nó inferior ao R23, por isso a tensão sobre R13, chamada internamente de V3 e V4, é menor que a tensão V2.

O pino R03 está conectado ao resistor Rx, que possui a função de ajuste de contraste do *display*. Quanto menor esse resistor maior o contraste do *display*. Esse nó origina a tensão chamada V5.

Tipicamente, temos R03 igual a 0 Ohms e R13 = R23 = R33 = 680KΩ, no entanto esses resistores podem ter valores entre 100K e 1MΩ de acordo com o *display* utilizado.

Existem ao todo quatro modos de funcionamento, que se diferenciam de acordo com o número de *back-planes* do *display*, como veremos mais adiante.

Os circuitos de temporização e de multiplexação analógica também fornecem os sinais necessários para o circuito excitador dos segmentos. Esse circuito utiliza os dados provenientes da memória de segmentos do controlador (20 *bytes*, localizados entre os endereços 0x0091 e 0x00A4 e nomeados de LCDM1 a LCDM20) para gerar os sinais de controle para os segmentos. A memória LCDM1 está associada aos pinos S0 e S1, LCDM2 está associada aos pinos S2 e S3 e assim por diante, até a memória LCDM20 que está associada aos pinos S38 e S39 (quando existentes). Um *bit* setado na memória indica que o segmento deve acender, enquanto um *bit* apagado indica que o segmento deve permanecer apagado.

A quantidade de segmentos disponíveis em cada plano varia de acordo com o modelo do *chip*: nos modelos F41x e FW42x, estão disponíveis somente os segmentos S0 a S23, nos modelos F42x, FE42x, F43x e FG43x, estão disponíveis somente os segmentos S0 a S31 e nos modelos F43x (100 pinos) e F44x, todos os 40 segmentos estão disponíveis.

O módulo possui ainda um controle de ativação/desativação (*bit* LCDCTL:LCDON) e um controle de ativação/desativação dos drivers de segmento (*bit* LCDCTL:LCDSON).

O *bit* LCDON é utilizado para a ativação e desativação do módulo, de forma a reduzir o consumo de corrente quando o módulo não está sendo utilizado.

O *bit* LCDSON é utilizado para controlar a ativação e desativação dos segmentos do LCD e pode ser utilizado quando se deseja apagar todo o *display* (sem apagar o conteúdo da memória de apresentação) ou fazê-lo piscar.

5.17.1. Modo Estático

Nesse modo, existe somente um plano de fundo, que é excitado pelo sinal COM0. Cada pino de segmento (S0 a Sxx) controla apenas um segmento do *display*.

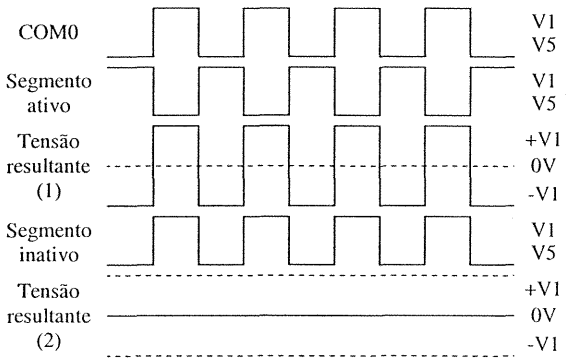


Figura 5-48

Quando está ativo, o segmento é alimentado com uma tensão igual a $V_{COM0} - V_{SEG}$, originando a tensão (1) na figura 5-48. Repare que o sinal COM0 e o sinal do segmento possuem fases opostas.

Quando o segmento está inativo, é alimentado por uma tensão igual a $V_{COM0} - V_{SEG}$, mas como ambos os sinais possuem a mesma fase, a diferença entre ambos é igual a zero, fazendo com que o segmento se apague (tensão (2) na figura 5-48).

O modo estático é selecionado quando os *bits* LCDCTL:LCDMx = 00. O mapeamento dos pinos de acionamento dos segmentos na memória do controlador LCD é o seguinte:

Endereço	Nome	*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0091	LCDM1	-	-	-	S1	-	-	-	S0
0x0092	LCDM2	-	-	-	S3	-	-	-	S2
0x0093	LCDM3	-	-	-	S5	-	-	-	S4
0x0094	LCDM4	-	-	-	S7	-	-	-	S6
0x0095	LCDM5	-	-	-	S9	-	-	-	S8
0x0096	LCDM6	-	-	-	S11	-	-	-	S10
0x0097	LCDM7	-	-	-	S13	-	-	-	S12
0x0098	LCDM8	-	-	-	S15	-	-	-	S14
0x0099	LCDM9	-	-	-	S17	-	-	-	S16
0x009A	LCDM10	-	-	-	S19	-	-	-	S18
0x009B	LCDM11	-	-	-	S21	-	-	-	S20
0x009C	LCDM12	-	-	-	S23	-	-	-	S22
0x009D	LCDM13	-	-	-	S25	-	-	-	S24
0x009E	LCDM14	-	-	-	S27	-	-	-	S26
0x009F	LCDM15	-	-	-	S29	-	-	-	S28
0x00A0	LCDM16	-	-	-	S31	-	-	-	S30
0x00A1	LCDM17	-	-	-	S33	-	-	-	S32
0x00A2	LCDM18	-	-	-	S35	-	-	-	S34
0x00A3	LCDM19	-	-	-	S37	-	-	-	S36
0x00A4	LCDM20	-	-	-	S39	-	-	-	S38

Tabela 5-44

5.17.2. Modo 2MUX

Nesse modo, temos dois planos de fundo, acionados pelos sinais COM0 e COM1 alternadamente.

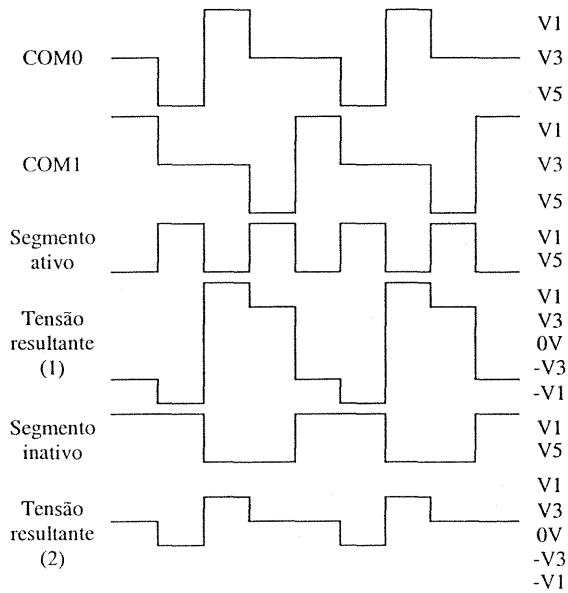


Figura 5-49

Quando um segmento está ativo, a tensão sobre ele é igual a VCOM0-VSEG ou VCOM1-VSEG (dependendo do plano em que está o segmento). A tensão resultante (1) demonstra um segmento ativo no plano COM0, a tensão resultante (2) representa um segmento inativo no plano COM1.

Esse modo de funcionamento é selecionado quando os *bits* LCDCTL:LCDMx = 01. O mapeamento dos pinos de acionamento dos segmentos na memória do controlador LCD é o seguinte:

Endereço	Nome	*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0091	LCDM1	-	-	S41	S1	-	-	S40	S0
0x0092	LCDM2	-	-	S43	S3	-	-	S42	S2
0x0093	LCDM3	-	-	S45	S5	-	-	S44	S4
0x0094	LCDM4	-	-	S47	S7	-	-	S46	S6
0x0095	LCDM5	-	-	S49	S9	-	-	S48	S8
0x0096	LCDM6	-	-	S51	S11	-	-	S50	S10
0x0097	LCDM7	-	-	S53	S13	-	-	S52	S12
0x0098	LCDM8	-	-	S55	S15	-	-	S54	S14
0x0099	LCDM9	-	-	S57	S17	-	-	S56	S16
0x009A	LCDM10	-	-	S59	S19	-	-	S58	S18
0x009B	LCDM11	-	-	S61	S21	-	-	S60	S20
0x009C	LCDM12	-	-	S63	S23	-	-	S62	S22
0x009D	LCDM13	-	-	S65	S25	-	-	S64	S24

Endereço	Nome	*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x009E	LCDM14	-	-	S67	S27	-	-	S66	S26
0x009F	LCDM15	-	-	S69	S29	-	-	S68	S28
0x00A0	LCDM16	-	-	S71	S31	-	-	S70	S30
0x00A1	LCDM17	-	-	S73	S33	-	-	S72	S32
0x00A2	LCDM18	-	-	S75	S35	-	-	S74	S34
0x00A3	LCDM19	-	-	S77	S37	-	-	S76	S36
0x00A4	LCDM20	-	-	S79	S39	-	-	S78	S38

Tabela 5-45

5.17.3. Modo 3MUX

No modo 3MUX, temos três planos de fundo que são ativados alternadamente pelos sinais COM0, COM1 e COM2.

Esse modo de funcionamento é selecionado quando os *bits* LCDCTL:LCDMx = 10. O mapeamento dos pinos de acionamento dos segmentos na memória do controlador LCD é o seguinte:

Endereço	Nome	*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0091	LCDM1	-	S81	S41	S1	-	S80	S40	S0
0x0092	LCDM2	-	S83	S43	S3	-	S82	S42	S2
0x0093	LCDM3	-	S85	S45	S5	-	S84	S44	S4
0x0094	LCDM4	-	S87	S47	S7	-	S86	S46	S6
0x0095	LCDM5	-	S89	S49	S9	-	S88	S48	S8
0x0096	LCDM6	-	S91	S51	S11	-	S90	S50	S10
0x0097	LCDM7	-	S93	S53	S13	-	S92	S52	S12
0x0098	LCDM8	-	S95	S55	S15	-	S94	S54	S14
0x0099	LCDM9	-	S97	S57	S17	-	S96	S56	S16
0x009A	LCDM10	-	S99	S59	S19	-	S98	S58	S18
0x009B	LCDM11	-	S101	S61	S21	-	S100	S60	S20
0x009C	LCDM12	-	S103	S63	S23	-	S102	S62	S22
0x009D	LCDM13	-	S105	S65	S25	-	S104	S64	S24
0x009E	LCDM14	-	S107	S67	S27	-	S106	S66	S26
0x009F	LCDM15	-	S109	S69	S29	-	S108	S68	S28
0x00A0	LCDM16	-	S111	S71	S31	-	S110	S70	S30
0x00A1	LCDM17	-	S113	S73	S33	-	S112	S72	S32
0x00A2	LCDM18	-	S115	S75	S35	-	S114	S74	S34
0x00A3	LCDM19	-	S117	S77	S37	-	S116	S76	S36
0x00A4	LCDM20	-	S119	S79	S39	-	S118	S78	S38

Tabela 5-46

5.17.4. Modo 4MUX

Nesse modo, temos quatro planos de fundo que são ativados alternadamente pelos sinais COM0, COM1, COM2 e COM3.

Esse modo de funcionamento é selecionado quando os *bits* LCDCTL:LCDMx = 11. O mapeamento dos pinos de acionamento dos segmentos na memória do controlador LCD é o seguinte:

Endereço	Nome	*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0091	LCDM1	S121	S81	S41	S1	S120	S80	S40	S0
0x0092	LCDM2	S123	S83	S43	S3	S122	S82	S42	S2
0x0093	LCDM3	S125	S85	S45	S5	S124	S84	S44	S4
0x0094	LCDM4	S127	S87	S47	S7	S126	S86	S46	S6
0x0095	LCDM5	S129	S89	S49	S9	S128	S88	S48	S8
0x0096	LCDM6	S131	S91	S51	S11	S130	S90	S50	S10
0x0097	LCDM7	S133	S93	S53	S13	S132	S92	S52	S12
0x0098	LCDM8	S135	S95	S55	S15	S134	S94	S54	S14
0x0099	LCDM9	S137	S97	S57	S17	S136	S96	S56	S16
0x009A	LCDM10	S139	S99	S59	S19	S138	S98	S58	S18
0x009B	LCDM11	S141	S101	S61	S21	S140	S100	S60	S20
0x009C	LCDM12	S143	S103	S63	S23	S142	S102	S62	S22
0x009D	LCDM13	S145	S105	S65	S25	S144	S104	S64	S24
0x009E	LCDM14	S147	S107	S67	S27	S146	S106	S66	S26
0x009F	LCDM15	S149	S109	S69	S29	S148	S108	S68	S28
0x00A0	LCDM16	S151	S111	S71	S31	S150	S110	S70	S30
0x00A1	LCDM17	S153	S113	S73	S33	S152	S112	S72	S32
0x00A2	LCDM18	S155	S115	S75	S35	S154	S114	S74	S34
0x00A3	LCDM19	S157	S117	S77	S37	S156	S116	S76	S36
0x00A4	LCDM20	S159	S119	S79	S39	S158	S118	S78	S38

Tabela 5-47

5.17.5. Conexões do Controlador de LCD

O controlador de LCD pode utilizar os seguintes pinos:

Sinal	Pino	Sinal	Pino
S0	P5.1	S24	P3.7 (modelos 43x 80 pinos) S24 (modelos 43x e 44x 100 pinos)
S1	P5.0	S25	P3.6 (modelos 43x 80 pinos) S25 (modelos 43x e 44x 100 pinos)
S2	P4.7 (modelos 41x e 43x 80 pinos)	S26	P3.5 (modelos 43x 80 pinos) S26 (modelos 43x e 44x 100 pinos)
S3	P4.6 (modelos 41x e 43x 80 pinos)	S27	P3.4 (modelos 43x 80 pinos) S27 (modelos 43x e 44x 100 pinos)
S4	P4.5 (modelos 41x e 43x 80 pinos)	S28	P3.3 (modelos 43x 80 pinos) S28 (modelos 43x e 44x 100 pinos)

Sinal	Pino	Sinal	Pino
S5	P4.4 (modelos 41x e 43x 80 pinos)	S29	P3.2 (modelos 43x 80 pinos) S29 (modelos 43x e 44x 100 pinos)
S6	P4.3 (modelos 41x e 43x 80 pinos)	S30	P3.1 (modelos 43x 80 pinos) S30 (modelos 43x e 44x 100 pinos)
S7	P4.2 (modelos 41x e 43x 80 pinos)	S31	P3.0 (modelos 43x 80 pinos)
S8	P4.1 (modelos 41x e 43x 80 pinos)	S32	S32 (modelos 43x e 44x 100 pinos)
S9	P4.0 (modelos 41x e 43x 80 pinos)	S33	S33 (modelos 43x e 44x 100 pinos)
S10	P3.7 (modelos 41x)	S34	P4.7 (modelos 43x e 44x 100 pinos)
S11	P3.6 (modelos 41x)	S35	P4.6 (modelos 43x e 44x 100 pinos)
S12	P3.5 (modelos 41x)	S36	P4.5 (modelos 43x e 44x 100 pinos)
S13	P3.4 (modelos 41x)	S37	P4.4 (modelos 43x e 44x 100 pinos)
S14	P3.3 (modelos 41x)	S38	P4.3 (modelos 43x e 44x 100 pinos)
S15	P3.2 (modelos 41x)	S39	P4.2 (modelos 43x e 44x 100 pinos)
S16	P3.1 (modelos 41x)	COM0	COM0
S17	P3.0 (modelos 41x)	COM1	P5.2
S18	P2.7 (modelos 41x e 43x 80 pinos) S18 (modelos 43x e 44x 100 pinos)	COM2	P5.3
S19	P2.6 (modelos 41x e 43x 80 pinos) S19 (modelos 43x e 44x 100 pinos)	COM3	P5.4
S20	P2.5 (modelos 41x)	R33	P5.7
S21	P2.4 (modelos 41x)	R23	P5.6
S22	P2.3 (modelos 41x)	R13	P5.5
S23	P2.2 (modelos 41x)	R03	R03

Tabela 5-48

5.17.6. Registradores do Controlador de LCD

O módulo LCD utiliza 21 registradores para o seu funcionamento:

- **LCDCTL** - controle do módulo;
- **LCDMEM1** a **LCDMEM20** - memórias de segmentos.

5.17.6.1. LCDCTL

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0090	LCDCTL	Leitura Escrita	LCDPx			LCDMXx		LCDSON	Não utilizado	LCDON
		Reset	0	0	0	0	0	0	0	0

LCDPx - seleção da função LCD para os pinos com função multiplexada LCD e E/S:
000 - pinos na sua função normal (símbolo **LCDSG0**);
001 - pinos de S0 a S15 na função LCD (símbolo **LCDSG0_1**);
010 - pinos de S0 a S19 na função LCD (símbolo **LCDSG0_2**);
011 - pinos de S0 a S23 na função LCD (símbolo **LCDSG0_3**);
100 - pinos de S0 a S27 na função LCD (símbolo **LCDSG0_4**);
101 - pinos de S0 a S31 na função LCD (símbolo **LCDSG0_5**);

110 - pinos de S0 a S35 na função LCD (símbolo **LCDSG0_6**);
111 - pinos de S0 a S39 na função LCD (símbolo **LCDSG0_7**).

LCDMXx - taxa de multiplexação do LCD:
00 - estático (símbolo **LCDSTATIC**);
01 - multiplexação de dois *backplanes* (símbolo **LCD2MUX**);
10 - multiplexação de três *backplanes* (símbolo **LCD3MUX**);
11 - multiplexação de quatro *backplanes* (símbolo **LCD4MUX**).

LCDSON - liga/desliga os segmentos do LCD, é uma forma rápida e prática de fazer o *display* piscar, sem ter de desativar o temporizador do LCD e nem o gerador de tensão:
0 - todos os segmentos do LCD permanecem desligados;
1 - os segmentos ativos nos registradores LCDMx são ativados no LCD (símbolo **LCDSON**).

LCDON - liga/desliga do temporizador do LCD e do gerador de tensão:
0 - temporizador do LCD e gerador de tensão desligados (LCD desligado);
1 - temporizador do LCD e gerador de tensão ligados (LCD ligado) (símbolo **LCDON**).

5.17.6.2. LCDMx

As memórias de segmento não são alteradas por um *reset* e o seu estado inicial é indefinido. Cada memória está associada fisicamente a dois pinos do microcontrolador:

Endereço	Nome	Pinos associados	
0x0091	LCDM1	S1	S0
0x0092	LCDM2	S3	S2
0x0093	LCDM3	S5	S4
0x0094	LCDM4	S7	S6
0x0095	LCDM5	S9	S8
0x0096	LCDM6	S11	S10
0x0097	LCDM7	S13	S12
0x0098	LCDM8	S15	S14
0x0099	LCDM9	S17	S16
0x009A	LCDM10	S19	S18

Endereço	Nome	Pixos associados	
0x009B	LCDM11	S21	S20
0x009C	LCDM12	S23	S22
0x009D	LCDM13	S25	S24
0x009E	LCDM14	S27	S26
0x009F	LCDM15	S29	S28
0x00A0	LCDM16	S31	S30
0x00A1	LCDM17	S33	S32
0x00A2	LCDM18	S35	S34
0x00A3	LCDM19	S37	S36
0x00A4	LCDM20	S39	S38

Tabela 5-49

5.17.7. Exemplo de Utilização

A seguir temos um exemplo de utilização do *display* LCD que acompanha a placa Microlab X1. Na versão 1.4 da placa ele se encontra conectado conforme a figura seguinte.

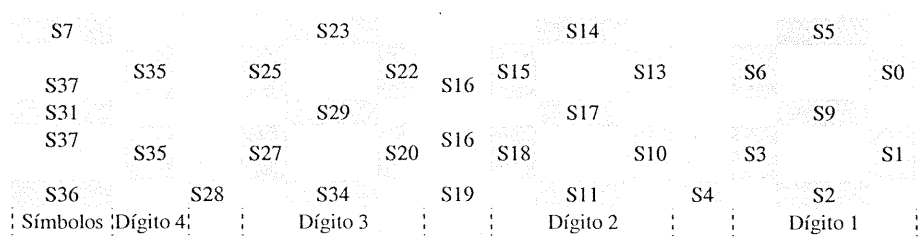


Figura 5-50

```
//*********************************************************************
// Exemplo de utilização do controlador de LCD
//*********************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*********************************************************************
// Este programa implementa um simples contador crescente que apre-
// senta os resultados em um display LCD estático
// Foi utilizado o hardware da placa Microlab X1
//*********************************************************************

#include "io430x44x.h"
#include "intrinsics.h"

// definições dos segmentos
#define seg0 LCDM1|=0x01;
#define seg1 LCDM1|=0x10;
#define seg2 LCDM2|=0x01;
#define seg3 LCDM2|=0x10;
#define seg4 LCDM3|=0x01;
#define seg5 LCDM3|=0x10;
#define seg6 LCDM4|=0x01;
#define seg7 LCDM4|=0x10;
#define seg8 LCDM5|=0x01;
#define seg9 LCDM5|=0x10;
#define seg10 LCDM6|=0x01;
#define seg11 LCDM6|=0x10;
#define seg12 LCDM7|=0x01;
#define seg13 LCDM7|=0x10;
#define seg14 LCDM8|=0x01;
#define seg15 LCDM8|=0x10;
#define seg16 LCDM9|=0x01;
#define seg17 LCDM9|=0x10;
#define seg18 LCDM10|=0x01;
#define seg19 LCDM10|=0x10;
#define seg20 LCDM11|=0x01;
#define seg21 LCDM11|=0x10;
#define seg22 LCDM12|=0x01;
#define seg23 LCDM12|=0x10;
#define seg24 LCDM13|=0x01;
#define seg25 LCDM13|=0x10;
#define seg26 LCDM14|=0x01;
#define seg27 LCDM14|=0x10;
#define seg28 LCDM15|=0x01;
#define seg29 LCDM15|=0x10;
#define seg30 LCDM16|=0x01;
#define seg31 LCDM16|=0x10;
#define seg32 LCDM17|=0x01;
#define seg33 LCDM17|=0x10;
#define seg34 LCDM18|=0x01;
#define seg35 LCDM18|=0x10;
#define seg36 LCDM19|=0x01;
#define seg37 LCDM19|=0x10;
#define seg38 LCDM20|=0x01;
#define seg39 LCDM20|=0x10;
#define seg_over seg7
#define seg_bat seg36
#define seg_menos seg31
#define seg_mais seg37
#define seg_dois_pontos seg16
#define seg_p3 seg28
#define seg_p2 seg19
#define seg_p1 seg4

unsigned char a,b,c,d;
```

```
#define BTIP0 1
#define BTIP1 2
#define BTIP2 4
//*********************************************************************
// Função: mostra_lcd
//*********************************************************************
// Entrada:    unsigned char digito4
//             unsigned char digito3
//             unsigned char digito2
//             unsigned char digito1
// Saída:      void
//*********************************************************************
// Esta função decodifica os quatro digitos recebidos e os apresenta
// no display LCD estático
//*********************************************************************
void mostra_lcd(unsigned char digito4, unsigned char digito3,
             unsigned char digito2, unsigned char digito1)
{
   // apaga todos os segmentos
   LCDM1 = LCDM2 = LCDM3 = LCDM4 = LCDM5 = LCDM6 = LCDM7 = 0;
   LCDM8 = LCDM9 = LCDM10 = LCDM11 = LCDM12 = LCDM13 = LCDM14 = 0;
   LCDM15 = LCDM16 = LCDM17 = LCDM18 = LCDM19 = LCDM20 = 0;
   // Os switch abaixo verificam o valor de cada dígito e executam as
   // macros correspondentes aos segmentos que devem ser acesos
   switch (digito1)
   {
      case 0: seg0; seg1; seg2; seg3; seg6; seg5; break;
      case 1: seg0; seg1; break;
      case 2: seg5; seg0; seg9; seg3; seg2; break;
      case 3: seg5; seg0; seg1; seg2; seg9; break;
      case 4: seg0; seg1; seg9; seg6; break;
      case 5: seg5; seg6; seg9; seg1; seg2; break;
      case 6: seg6; seg3; seg2; seg1; seg9; seg5; break;
      case 7: seg5; seg0; seg1; break;
      case 8: seg0; seg1; seg2; seg3; seg6; seg5; seg9; break;
      case 9: seg5; seg0; seg1; seg9; seg6; seg2;      break;
   }
   switch (digito2)
   {
      case 0: seg13; seg10; seg11; seg18; seg15; seg14; break;
      case 1: seg13; seg10; break;
      case 2: seg14; seg13; seg17; seg18; seg11; break;
      case 3: seg14; seg13; seg17; seg10; seg11; break;
      case 4: seg15; seg17; seg13; seg10; break;
      case 5: seg14; seg15; seg17; seg10; seg11; break;
      case 6: seg14; seg15; seg17; seg10; seg11; seg18; break;
      case 7: seg14; seg13; seg10; break;
      case 8: seg17; seg18; seg11; seg10; seg13; seg14; seg15; break;
      case 9: seg11; seg10; seg13; seg14; seg15; seg17; break;
   }
   switch (digito3)
   {
      case 0: seg23; seg22; seg20; seg34; seg27; seg25; break;
      case 1: seg22; seg20; break;
      case 2: seg23; seg22; seg29; seg27; seg34; break;
      case 3: seg23; seg22; seg29; seg20; seg34; break;
      case 4: seg25; seg29; seg22; seg20; break;
      case 5: seg23; seg25; seg29; seg20; seg34; break;
      case 6: seg23; seg25; seg29; seg20; seg34; seg27; break;
      case 7: seg23; seg22; seg20; break;
      case 8: seg23; seg22; seg20; seg34; seg27; seg25; seg29; break;
      case 9: seg20; seg22; seg23; seg25; seg29; break;
   }
   if (digito4) seg35;
}
```

```
//*********************************************************************
// Função: trata_BT
//*********************************************************************
// Esta função trata a interrupção do contador 2 do timer básico. O
// mesmo está configurado para dividir o ACLK por 256 * 128 = 32768
// , resultando em 1 interrupção por segundo (quando utilizando um
// cristal de 32768Hz no LFXT1).
//*********************************************************************
#pragma vector = BASICTIMER_VECTOR
__interrupt void trata_BT(void)
{
   a++;        // incrementa a
   if (a>9) // se a>9
   {
      a=0;     // zera a
      b++;     // e incrementa b
      if (b>9) // se b>9
      {
         b=0;     // zera b
         c++;     // e incrementa c
         if (c>9) // se c>9
         {
            c=0;          // zera c
            d++;          // e incrementa d
            if (d>1) d=0; // se d>1, zera d
         }
      }
   }
   mostra_lcd(d,c,b,a);
   // um segmento qualquer pode ser aceso, bastando executar
   // a macro correspondente:
   seg_over;
}

void main(void)
{
   WDTCTL = WDTPW+WDTHOLD;        // desliga o watchdog
   // configura o FLL para 1MHz (a partir do LFXT1 a 32768Hz)
   SCFI0 = FLLD_1;
   SCFQCTL = 0x1E;
   FLL_CTL0 = DCOPLUS;
   FLL_CTL1 = SMCLKOFF + XT2OFF;
   while (!IFG1_bit.OFIFG) IFG1_bit.OFIFG = 0;
   // configura o timer básico
   BTCTL = BTDIV + BT_fCLK2_DIV128;     // ACLK / (256 * 128)
   IE2 = BTIE;
   // liga o LCD, seleciona o modo estático
   // habilita todas as saídas do módulo LCD
   LCDCTL = LCDON+LCDSTATIC+LCDP2+LCDP1+LCDP0;
   // configura todos os pinos como saída, para reduzir o consumo
   P1DIR = P2DIR = P3DIR = P4DIR = P5DIR = P6DIR = 0xFF;
   // inicia em zero a contagem
   a = b = c = d = 0;
   __enable_interrupt(); // habilita as interrupções
   __low_power_mode_3(); // entrada no modo LMP3
   while (1);            // aguarda a interrupção
}
```

Exemplo 5-15

A corrente total drenada pelo circuito é de aproximadamente 14μA/h, o que significa que com o uso de uma pilha do tipo CR2032, com capacidade de 280mA/h, o circuito pode funcionar por cerca de 20.000 horas, ou 833,33 dias, ou seja, mais de dois anos!

5.18. Multiplicador por *Hardware*

Alguns modelos MSP430 incluem também um periférico especial dedicado a operações de cálculos matemáticos, mais especificamente multiplicações.

O *hardware* multiplicador possui as seguintes características:

- Independente da CPU;
- Capaz de executar multiplicações inteiras sinalizadas e não-sinalizadas;
- Capaz de executar operações de multiplicação e acúmulo (MAC) sinalizadas e não--sinalizadas;
- Pode realizar operações envolvendo operandos de 8 e/ou 16 *bits* (8 x 8, 8 x 16, 16 x 8 e 16 x 16), contando com um acumulador de 32 *bits*;
- Operações em três ciclos MCLK.

O funcionamento do multiplicador é muito simples: o *chip* contém um circuito multiplicador de 16 x 16 *bits* que é alimentado por dois operandos chamados OP1 e OP2. O conteúdo do primeiro operando é obtido de um dos seguintes registradores:

- MPY - para operações de multiplicação sem sinal;
- MPYS - para operações de multiplicação sinalizadas;
- MAC - para operações de multiplicação e acúmulo sem sinal;
- MACS - para operações de multiplicação e acúmulo sinalizadas.

Dependendo do registrador em que se escreve o operando OP1, selecionamos o tipo de operação a ser realizado.

O conteúdo do segundo operando é diretamente escrito no registrador OP2. Esta operação de escrita inicia também o processo de cálculo do circuito multiplicador.

Após três ciclos do *clock* principal, o resultado da operação está disponível nos registradores RESLO (parte menos significativa do resultado), RESHI (parte mais significativa do resultado) e SUMEXT (extensão do sinal do resultado).

Uma nova escrita no registrador OP2 inicia um novo processo de multiplicação utilizando o conteúdo previamente escrito no OP1.

Observe que nas operações que envolvem operandos sinalizados, eles devem estar no formado de complemento de dois. Neste caso, os registradores de resultado também são formatados segundo este padrão. O resultado armazenado no registrador SUMEXT vai depender do tipo de operação realizado:

Operação	Resultado no SUMEXT
Multiplicação não sinalizada	O seu conteúdo é sempre 0x0000
Multiplicação sinalizada	O conteúdo do SUMEXT é a extensão do sinal do resultado: - 0x0000 se o resultado foi positivo ou zero - 0xFFFF se o resultado foi negativo
MAC não-sinalizada	O seu conteúdo é igual ao transporte da operação MAC: - 0x0000 se não houve transporte - 0x0001 se houve transporte
MAC sinalizada	O conteúdo do SUMEXT é a extensão do sinal do resultado: - 0x0000 se o resultado foi positivo ou zero - 0xFFFF se o resultado foi negativo

Tabela 5-50

Um detalhe importante a ser observado é que nas operações MAC sinalizadas não há sinalização de condições como *overflow* (quando a soma de dois números positivos resulta um número maior que 0x7FFFFFFF) ou *underflow* (quando a soma de dois números negativos resulta um número menor que 0xFFFFFFFF). Em ambos os casos, a representação do número ultrapassa a sua faixa de representação e invade o domínio do conjunto oposto.

Nesse tipo de operação, o registrador SUMEXT sempre representa corretamente o sinal do resultado da operação, mas o *software* do usuário deve providenciar a correção necessária ao processamento do resultado.

O multiplicador por *hardware* pode ser encontrado nos MSP430F14x, F16x, FE41x e F44x.

5.18.1. Registradores

5.18.1.1. MPY

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x0130	MPY	Leitura / Escrita	Operando 1 para multiplicações sem sinal				
		Reset	?	?	...	?	?

5.18.1.2. MPYS

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x0132	MPYS	Leitura / Escrita	Operando 1 para multiplicações sinalizadas				
		Reset	?	?	...	?	?

5.18.1.3. MAC

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x0134	MAC	Leitura / Escrita	Operando 1 para MAC sem sinal				
		Reset	?	?	...	?	?

5.18.1.4. MACS

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x0136	MACS	Leitura / Escrita	Operando 1 para MAC sinalizada				
		Reset	?	?	...	?	?

5.18.1.5. OP2

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x0138	OP2	Leitura Escrita	Operando 2 da operação de multiplicação. A escrita nesse registrador inicia a operação de multiplicação				
		Reset	?	?	...	?	?

5.18.1.6. RESLO

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x013A	RESLO	Leitura Escrita	Parte menos significativa do resultado				
		Reset	?	?	...	?	?

5.18.1.7. RESHI

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x013C	RESHI	Leitura Escrita	Parte mais significativa do resultado				
		Reset	?	?	...	?	?

5.18.1.8. SUMEXT

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x013E	SUMEXT	Leitura Escrita	Extensão do sinal do resultado				
		Reset	?	?	...	?	?

5.18.2. Exemplos de Utilização

O multiplicador por *hardware* é automaticamente utilizado pelo compilador, desde que a opção *Hardware Multiplier* esteja habilitada (opções do projeto, aba *Target* dentro das opções gerais (*General Options*)).

Assim, mesmo num programa tão simples como:

```
#include <io430x14x.h>

void main(void)
{
  volatile unsigned int va, vb, vc;
  va = 1200;
  vb = 35;
  vc = va * vb;
}
```

Caso a opção esteja selecionada, o compilador vai identificar a operação de multiplicação e utilizar o *hardware* de multiplicação interno.

Uma outra possibilidade que vamos demonstrar é a utilização do *hardware* de multiplicação controlado diretamente pelo programa em C. No próximo exemplo, uma matriz de 15 elementos do tipo inteiro sinalizado será mutiplicada por uma constante qualquer (no caso 5).

```
//*******************************************************************
// Exemplo de multiplicação por hardware
//*******************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*******************************************************************
// Este programa multiplica os elementos de uma matriz pelo valor
// 5 e armazena novamente na matriz. A operação é feita utilizando
// o multiplicador por hardware
//*******************************************************************

#include <io430x14x.h>

volatile int matriz[15] =
  {
        200, -50, 920, 333, 477,
        11500, 25, -670, -344, 3517,
        9999, 2000, 555, 190, -2345
  };

void main( void )
{
  unsigned char temp;
  // carrega a constante no registrador MPYS
  // isto seleciona uma multiplicação sinalizada
  MPYS = 5;
  for (temp=0; temp<15; temp++)
  {
        // carrega o valor da matriz no operando 2 (OP2)
        // iniciando a multiplicação
        OP2 = matriz[temp];
        // escreve o resultado de volta na matriz
        matriz[temp]=RESLO;
  }
  while (1);
}
```

Exemplo 5-16

5.19. Controlador de DMA

Alguns modelos de MSP430 (15x, 16x e FG43x) incluem um controlador de acesso direto a memória (DMA) capaz de automatizar operações de transferência de informação dentro do espaço de endereçamento da memória.

Basicamente, o que o controlador de DMA faz é transferir informação de um endereço fonte para um endereço destino, de forma autônoma, sem intervenção da CPU e sem a necessidade de se escrever código para efetuar essa operação (a não ser o necessário para a configuração do DMA, é claro).

As transferências ocorrem de forma independente da CPU, mas como o controlador de DMA acessa a mesma memória que a CPU, durante uma operação de transferência, a CPU é paralisada, voltando ao seu estado normal ao término da operação. É possível programar o controlador de DMA para iniciar uma transferência a partir de diversos eventos de *hardware* e de *software*, como veremos mais adiante.

A transferência de um *byte* ou *word* ocorre em dois ciclos do *clock* principal (MCLK), como veremos mais adiante.

Nos modelos F15x e F16x, estão disponíveis três canais independentes de DMA e nos modelos FG43x, apenas um canal está disponível.

As aplicações do DMA são tão interessantes quanto vastas, permitindo a automatização de diversas operações, tais como: a movimentação automática dos dados recebidos da UART para um *buffer* na memória, dos dados convertidos pelo ADC para a memória, de um bloco de dados da memória para o DAC, chegando a aplicações complexas como a implementação de filtros digitais sem praticamente nenhuma intervenção da CPU (como na nota de aplicação SLAA228 da *Texas Instruments* (referência 20), que demonstra a criação de filtros FIR utilizando apenas recursos de *hardware* como o *timer*, ADC, DMA e *hardware* de multiplicação).

5.19.1. Modos de Operação e Endereçamento

Cada canal de DMA pode operar em nove modos de endereçamento e seis modos de transferência.

Os modos de endereçamento possíveis são controlados pelos *bits* DMASRCINCRx e DMADSTINCRx (ambos localizados no registrador DMAxCTL):

DMASRCINCRx (fonte)	DMADSTINCRx (destino)	Modo de Endereçamento
00 01	00 01	Endereço único para endereço único
00 01	10	Endereço único para bloco (o endereço de destino é decrementado a cada transferência)
00 01	11	Endereço único para bloco (o endereço de destino é incrementado a cada transferência)
10	00 01	Bloco de endereços para endereço único (o endereço fonte é decre-mentado a cada transferência)
11	00 01	Bloco de endereços para endereço único (o endereço fonte é incre-mentado a cada transferência)
10	10	Bloco de endereços para bloco de endereços (os endereços fonte e destino são decrementados a cada transferência)

DMASRCINCRx (fonte)	DMADSTINCRx (destino)	Modo de Endereçamento
10	11	Bloco de endereços para bloco de endereços (o endereço fonte é decrementado e o endereço de destino é incrementado a cada transferência)
11	10	Bloco de endereços para bloco de endereços (o endereço fonte é incrementado e o endereço de destino é decrementado a cada transferência)
11	11	Bloco de endereços para bloco de endereços (os endereços fonte e destino são incrementados a cada transferência)

Tabela 5-51

O tamanho dos blocos é especificado pelo registrador DMAxSZ. É possível transferir blocos de até 65.535 *bytes* ou *words*. No caso de transferências de bloco para bloco, ambos sempre vão possuir o mesmo tamanho, determinado pelo registrador DMAxSZ.

Os modos de endereçamento listados na tabela 5-51 admitem ainda a transferência de *bytes* ou *words*, selecionáveis para o endereço fonte e endereço destino:

- *Byte* para *byte*;
- *Byte* para *word* (o *byte* superior da *word* de destino é apagado);
- *Word* para *byte* (o *byte* de destino recebe o valor do *byte* menos significativo da *word* do endereço fonte);
- *Word* para *word*.

A seleção do tamanho do dado da fonte e do destino é realizada pelos *bits* DMASRCBYTE e DMADSTBYTE, localizados no registrador DMAxCTL.

Além dos modos de endereçamento apresentados, estão disponíveis também seis modos de transferência, selecionados pelos *bits* DMADTx (registrador DMAxCTL):

DMADTx	Modo de Transferência	Descrição
000	Transferência única	Cada evento de disparo do DMA provoca uma única transferência. Após a transferência dos "n" dados especificados no DMAxSZ, o canal de DMA é automaticamente desabilitado (DMAxCTL:DMAEN = 0).
001	Transferência de um bloco	Cada evento de disparo do DMA provoca a transferência de um bloco completo (cujo tamanho é especificado pelo registrador DMAxSZ). Após isso, o canal de DMA é automaticamente desabilitado (DMAxCTL:DMAEN = 0).
010 011	Transferência de um bloco em rajadas	Cada evento de disparo do DMA provoca a transferência de um bloco completo (cujo tamanho é especificado pelo registrador DMAxSZ). No modo de transferência em rajadas (burst), a CPU opera por dois ciclos MCLK a cada quatro transferências, resultando uma performance de 20% da velocidade máxima naquela freqüência. Após a transferência do bloco, o canal de DMA é automaticamente desabilitado (DMAxCTL:DMAEN = 0).
100	Transferência única repetitiva	Cada evento de disparo do DMA provoca uma única transferência. O canal de DMA permanece sempre habilitado (DMAxCTL:DMAEN = 1).
101	Transferência de um bloco repetitiva	Cada evento de disparo do DMA provoca a transferência de um bloco completo (cujo tamanho é especificado pelo registrador DMAxSZ). O canal de DMA permanece habilitado (DMAxCTL:DMAEN = 1).
110 111	Transferência de blocos, repetitiva, em rajadas	Um evento de disparo do DMA provoca a transferência de um bloco completo (cujo tamanho é especificado pelo registrador DMAxSZ). No modo de transferência em rajadas (burst), a CPU opera por dois ciclos MCLK a cada quatro transferências, resultando uma performance de 20% da velocidade máxima naquela freqüência. Após a transferência do primeiro bloco, o canal de DMA permanece ativado (DMAxCTL:DMAEN = 1) e uma nova transferência tem início automaticamente, independentemente de disparo.

Tabela 5-52

5.19.1.1. Transferência Única

No modo de transferência única, apenas um *byte/word* é transferido a cada evento de disparo do DMA.

A cada transferência o valor presente no registrador DMAxSZ é decrementado de um e os registradores DMAxSA (que aponta o endereço fonte) e DMAxDA (que aponta o endereço de destino) são modificados de acordo com o modo de endereçamento utilizado.

Durante a transferência, a execução do programa é paralisada. Caso o *bit* DMAONFETCH (registrador DMACTL1) esteja setado, a instrução em execução é completada e a transferência tem início antes da próxima busca de instrução. Caso o *bit* DMAONFETCH esteja apagado, a CPU é paralisada no instante em que o evento de disparo do DMA é recebido. Neste caso, a execução da instrução é completada após a transferência ser finalizada.

Uma vez que o registrador DMAxSZ atinja o valor zero, o canal de DMA é automaticamente desabilitado (no modo de transferência única não repetitiva). No caso do modo de transferência única repetitiva, ele permanece habilitado e um novo evento provoca o início de uma nova transferência.

Em ambos os casos, após a transferência da quantidade de *bytes/words* especificada pelo DMAxSZ, ele é carregado com o seu valor inicial.

Observe que caso o registrador DMAxSZ seja carregado com o valor zero e ocorra um evento de disparo do DMA, nenhuma transferência será realizada e o registrador DMAxSZ permanece com o valor zero.

5.19.1.2. Transferência de Bloco

No modo de transferência de bloco, um bloco com tamanho definido pelo registrador DMAxSZ é transferido completamente a cada evento de disparo do DMA.

A cada *byte/word* transferido o valor presente no registrador DMAxSZ é decrementado de um e os registradores DMAxSA e DMAxDA são modificados de acordo com o modo de endereçamento utilizado.

Durante a transferência, a execução do programa é paralisada. Caso o *bit* DMAONFETCH (registrador DMACTL1) esteja setado, a instrução em execução é completada e a transferência tem início antes da próxima busca de instrução. Caso o *bit* DMAONFETCH esteja apagado, a CPU é paralisada no instante em que o evento de disparo do DMA é recebido. Neste caso, a execução da instrução é completada após a transferência do bloco ser finalizada.

Uma vez que o registrador DMAxSZ atinja o valor zero, o canal de DMA é automaticamente desabilitado (no modo de transferência de bloco não repetitiva). No caso do modo de transferência de bloco repetitiva, ele permanece habilitado e um novo evento provoca o início de uma nova transferência de bloco.

Em ambos os casos, após a transferência do bloco, o registrador DMAxSZ é carregado com o seu valor inicial e o *flag* DMAIFG do canal é ativado. Se a interrupção do canal estiver habilitada (DMAxCTL:DMAIE = 1) e o controle global de interrupções também estiver habilitado (SR:GIE = 1), o fluxo de execução do programa será desviado para o vetor de interrupção correspondente.

Observe que, caso o registrador DMAxSZ seja carregado com o valor zero e ocorra um evento de disparo do DMA, nenhuma transferência será realizada e o registrador DMAxSZ permanece com o valor zero.

5.19.1.3. Transferência de Bloco em Rajadas

No modo de transferência de bloco em rajadas, um bloco com tamanho definido pelo registrador DMAxSZ é transferido completamente a cada evento de disparo do DMA. Nesse modo, a atividade da CPU é intercalada com a operação do DMA, resultando uma performance de 20% da velocidade normal da CPU. Da mesma forma, a operação de transferência ocorre a 80% da sua velocidade normal.

A cada *byte/word* transferido o valor presente no registrador DMAxSZ é decrementado de um e os registradores DMAxSA e DMAxDA são modificados de acordo com o modo de endereçamento utilizado.

Durante a transferência a execução do programa é paralisada, mas a cada quatro operações de transferência de *byte* ou *word*, a CPU é ativada por dois ciclos MCLK. Desta forma, a cada dez ciclos MCLK, oito são dedicados ao DMA e dois à CPU.

Caso o *bit* DMAONFETCH (registrador DMACTL1) esteja setado, a instrução em execução durante o evento de disparo do DMA é completada e a transferência tem início antes da próxima busca de instrução. Caso o *bit* DMAONFETCH esteja apagado, a CPU é paralisada no instante em que o evento de disparo do DMA é recebido.

Uma vez que o registrador DMAxSZ atinja o valor zero, o canal de DMA é automaticamente desabilitado (no modo de transferência de bloco em rajadas não repetitivo) e a execução do programa retorna à sua velocidade normal.

No caso do modo de transferência de bloco em rajadas repetitivo, ele permanece habilitado e uma nova transferência tem início imediatamente após o término da transferência corrente. Nenhum evento de disparo adicional é necessário. Nessas condições, a operação do DMA somente pode ser interrompida apagando o *bit* DMAxCTL:DMAEN ou por uma interrupção NMI (desde que o *bit* DMACTL1:ENNMI esteja setado).

Em ambos os casos, após a transferência do bloco, o registrador DMAxSZ é carregado com o seu valor inicial e o *flag* DMAIFG do canal é ativado. Se a interrupção do canal estiver habilitada (DMAxCTL:DMAIE = 1) e o controle global de interrupções também estiver habilitado (SR:GIE = 1), o fluxo de execução do programa será desviado para o vetor de interrupção correspondente.

Observe que, caso o registrador DMAxSZ seja carregado com o valor zero e ocorra um evento de disparo do DMA, nenhuma transferência será realizada e o registrador DMAxSZ permanece com o valor zero.

5.19.2. Eventos de Disparo do DMA

O início de uma transferência DMA depende da ocorrência de um evento de disparo (*trigger*).

A sensibilidade de disparo da operação de transferência pode ser tanto à transição quanto ao nível do sinal. Essa função é selecionada pelo bit DMALEVEL (registrador DMAxCTL).

No modo sensível a borda (DMALEVEL = 0), uma transição de nível lógico "0" para nível lógico "1" do sinal da fonte de disparo selecionada provoca o início da operação de transferência (de um *byte/word* ou de um bloco, dependendo do modo de transferência selecionado).

No modo sensível ao nível (DMALEVEL = 1), a transferência ocorre apenas enquanto o sinal de disparo permanecer em nível lógico "1". Se durante a operação o sinal voltar para nível "0", a operação de transferência é suspendida. Neste caso, quando o sinal voltar novamente ao nível "1", a operação continuará do ponto em que havia parado. Se durante o estado "suspenso" da

transferência o *software* alterar os registradores do controlador de DMA, a transferência é cancelada. Neste caso, quando o sinal de disparo transitar para nível "1", uma nova operação de DMA terá início.

O modo de disparo sensível ao nível é recomendado somente para o uso com o disparo externo (pino DMAE0).

O controlador de DMA permite selecionar um entre dezesseis eventos diferentes para provocar o disparo de uma operação de transferência. O evento de disparo é selecionado pelos *bits* DMAxTSELx localizados no registrador DMACTL0.

Fonte	DMAxTSELx	Descrição
Software	0000	Disparo por *software*. O início da transferência é comandado pelo *bit* DMAREQ (registrador DMAxCTL) que é automaticamente apagado pelo controlador após a transferência.
CCP2 *timer* A	0001	A transferência é iniciada quando o *flag* CCIFG do canal 2 do *timer* A é setado (desde que essa interrupção não esteja habilitada). O controlador de DMA apaga automaticamente o *flag*.
CCP2 *timer* B	0010	A transferência é iniciada quando o *flag* CCIFG do canal 2 do *timer* B é setado (desde que essa interrupção não esteja habilitada). O controlador de DMA apaga automaticamente o *flag*.
USART0 RX	0011	A transferência é iniciada quando a USART0 recebe um novo dado. No modo UART ou SPI, a transferência é iniciada quando o *flag* URXIFG0 é setado (desde que a interrupção URXIE0 esteja desabilitada). O controlador de DMA apaga automaticamente o *flag* ao iniciar a transferência. No modo I2C, a transferência tem início na recepção de dados (e não quando o *flag* RXRDYIFG é setado). Neste caso, o *flag* RXRDYIFG não é apagado pelo controlador de DMA.
USART0 TX	0100	A transferência é iniciada quando a USART0 está pronta para transmitir um novo dado. No modo UART ou SPI, a transferência é iniciada quando o *flag* UTXIFG0 é setado (desde que a interrupção UTXIE0 esteja desabilitada). O controlador de DMA apaga automaticamente o *flag* ao iniciar a transferência. No modo I2C, a transferência tem início na condição de transmissor pronto (e não quando o *flag* TXRDYIFG é setado). Neste caso, o *flag* TXRDYIFG não é apagado pelo controlador de DMA.
DAC0	0101	A transferência é iniciada quando o canal 0 do DAC está pronto para uma nova conversão (*flag* DAC12IFG = 1), desde que a interrupção DAC12IE esteja desabilitada. O *flag* DAC12IFG é automaticamente apagado pelo controlador de DMA no início da transferência.
ADC12	0110	A transferência é iniciada no término da conversão do ADC12 (*flag* ADC12IFGx = 1). Quando utilizado o modo de conversão de um canal, o *flag* daquele canal provoca o disparo. Quando utilizado o modo seqüencial, o *flag* do último canal da seqüência é o que provoca o disparo. Setar o *flag* por *software* não inicia uma transferência.
CCP0 *timer* A	0111	A transferência é iniciada quando o *flag* CCIFG do canal 0 do *timer* A é setado (desde que essa interrupção não esteja habilitada). O controlador de DMA apaga automaticamente o *flag* no início da transferência.
CCP0 *timer* B	1000	A transferência é iniciada quando o *flag* CCIFG do canal 2 do *timer* B é setado (desde que essa interrupção não esteja habilitada). O controlador de DMA apaga automaticamente o *flag* no início da transferência.
USART1 RX	1001	A transferência é iniciada quando a USART1 recebe um novo dado (*flag* URXIFG0 = 1), desde que a interrupção URXIE0 esteja desabilitada. O controlador de DMA apaga automaticamente o *flag* ao iniciar a transferência.
USART1 TX	1010	A transferência é iniciada quando a USART1 está pronta para transmitir um novo dado (*flag* UTXIFG0 = 1), desde que a interrupção UTXIE0 esteja desabilitada. O controlador de DMA apaga automaticamente o *flag* ao iniciar a transferência.
Multiplicador por *Hardware*	1011	A transferência é iniciada quanto o multiplicador está pronto para uma nova operação.
-	1100	Nenhuma transferência.
-	1101	Nenhuma transferência.
DMA	1110	A transferência é iniciada quando o *flag* DMAxIFG é setado. O canal 0 é disparado pelo canal 2, o canal 1 é disparado pelo 0 e o canal 2 é disparado pelo 1. O *flag* DMAxIFG não é apagado pelo controlador.
Sinal externo	1111	A transferência é iniciada pelo sinal externo DMAE0 (pino P2.6 nos modelos F15x e 16x e pino 3.6 nos modelos FG43x).

Tabela 5-53

A seleção de uma fonte de disparo do DMA deve ser realizada sempre com o canal desativado (DMAxCTL:DMAEN = 0), de forma a evitar um disparo indesejado do módulo.

Caso o disparo da fonte selecionada tenha ocorrido antes da sua seleção, nenhuma operação de DMA terá início. Somente um novo evento de disparo provoca o início da transferência de DMA.

Quando-se utiliza o DMA para a escrita na memória, o fabricante recomenda programar o bit DMAONFETCH em "1", de forma que o início da operação de transferência ocorra somente após o término da execução da instrução em curso. A desobediência a esta recomendação pode provocar resultados imprevistos na operação de escrita.

5.19.3. Encerrando uma Operação DMA

Como já vimos, uma operação de DMA suspende o funcionamento da CPU (a não ser nos modos de transferência em rajada).

Caso seja necessário cancelar uma operação de transferência executada pelo controlador de DMA, duas alternativas estão disponíveis:

- Interrupção NMI: caso o *bit* ENNMI esteja setado, uma interrupção NMI vai interromper e encerrar a transferência em andamento.
- Desabilitação do *bit* DMAEN: no caso dos modos de transferência em rajada, quando a CPU ainda permanece ativa, é possível abortar uma operação apagando o *bit* DMAEN.

5.19.4. Prioridades entre os Canais

Nos modelos com mais de um canal de DMA (15x e 16x), no caso da ocorrência de eventos de disparo em mais de um canal simultaneamente, ou ainda na hipótese do disparo de um canal enquanto outro está no meio de uma operação de transferência, existe um mecanismo de prioridade, em que o canal 0 possui a maior prioridade e o 2, a menor.

Desta forma, os canais 0 e o 1 forem disparados simultaneamente, o 0 será ativado primeiro e após completar a sua operação, o canal 1 será disparado.

Observe também que uma operação de transferência não é interrompida por outra, não importando a sua prioridade.

Alternativamente, é possível alterar a prioridade dos canais, utilizando o *bit* ROUNDROBIN (registrador DMACTL1). Quando esse *bit* está ativado (nível "1"), o sistema atua como uma espécie de fila, na qual o canal que completa uma transferência torna-se o de menor prioridade (final da fila) e os demais canais permanecem na sua ordem de prioridade original.

Assim, partindo da seqüência original 0-1-2 e supondo a ocorrência de um disparo do canal 0, a nova seqüência será: 1-2-0. Supondo um novo disparo do canal 2, a nova seqüência passará a ser 1-0-2 e assim por diante.

5.19.5. Interrupções e DMA

Uma característica importante das operações DMA é que elas interrompem a execução do programa em qualquer ponto em que ele se encontre. Isso inclui as rotinas de tratamento de interrupção.

No caso de RTIs críticas, pode ser necessário desabilitar os canais de DMA (DMAxCTL:DMAEN = 0) no início da execução da RTI e reabilitá-los ao seu término.

O controlador de DMA também pode gerar interrupções. Neste caso, o *flag* DMAIFG correspondente a cada canal (registrador DMAxCTL) é setado ao término da operação de transferência (DMAxSZ = 0). Caso o controle individual de interrupção do canal (DMAxCTL:DMAIE) e o controle global de interrupções (SR:GIE) estejam habilitados, o programa será desviado para o vetor de interrupção correspondente (ver a tabela 5-1 para detalhes sobre o vetor).

Os flags não são apagados automaticamente pelo hardware, sendo assim, o software do usuário deve providenciar o apagamento deles em algum ponto da RTI.

5.19.6. Características de Temporização das Transferências

Cada transferência (de *byte/word* ou de bloco) necessita de um ciclo MCLK inicial (para sincronização do controlador DMA), dois ciclos MCLK para a transferência de cada *byte* ou *word* e mais um ciclo MCLK de espera após o término da transferência (do *byte/word* ou do bloco). A transferência única de um *byte/word* necessita normalmente de quatro ciclos MCLK para ser completada.

Uma característica muito interessante do controlador de DMA reside no fato de que, apesar de utilizar o *clock* principal (MCLK), o módulo possui uma relativa independência do modo de operação da CPU.

Caso a fonte do MCLK esteja ativa (por exemplo, o oscilador LFXT1 esteja ativado), mas a CPU esteja num modo de baixa potência, o DMA precisará de dois ciclos MCLK para sincronização inicial antes do início da transferência. A transferência única de um *byte/word* nessas condições necessita de cinco ciclos MCLK para ser completada.

Se a fonte do MCLK estiver inativa (a CPU esteja num modo de baixa potência), o controlador de DMA tem a capacidade de ativar o oscilador DCO, que funcionará como fonte de *clock* durante o processo de transferência. Nesse caso, um tempo de 6µs (1µs nos modelos 2xx) para ativação do DCO é necessário para a sincronização inicial. Completada a transferência, o DCO é desativado. Nenhuma interferência da CPU é necessária e o estado dela não é alterado. A transferência única de um *byte/word* nessas condições necessita de cinco ciclos MCLK mais um tempo de 6µs (1µs nos modelos 2xx) para ser completada.

5.19.7. Conexões do Controlador de DMA

O controlador de DMA pode utilizar o seguinte pino:

Sinal	Pino
DMAE0	P3.6 (modelos FG43x) P2.6 (modelos 15x e 16x)

Tabela 5-54

5.19.8. Registradores do Controlador de DMA

O controlador de DMA utiliza os seguintes registradores para o seu funcionamento:

- **DMACTL0** - seleção das fontes de disparo de cada canal;
- **DMACTL1** - controle da operação do DMA;

- **DMA0CTL, DMA1CTL** e **DMA2CTL** - controle dos canais de DMA;
- **DMA0SA, DMA1SA** e **DMA2SA** - endereço fonte da transferência;
- **DMA0DA, DMA1DA** e **DMA2DA** - endereço destino da transferência;
- **DMA0SZ, DMA1SZ** e **DMA2SZ** - número de *bytes/words* a ser transferido.

5.19.8.1. DMACTL0

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0122	DMACTL0	Leitura / Escrita	Reservados				DMA2TSELx			
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura / Escrita	DMA1TSELx				DMA0TSELx			
		Reset	0	0	0	0	0	0	0	0

DMA2TSELx -

DMA1TSELx -

DMA0TSELx - seleção do disparador da transferência DMA:

0000 - *bit* DMAREQ (disparo por *software*) (símbolo **DMA0TSEL_0**, **DMA1TSEL_0** ou **DMA2TSEL_0**);

0001 - canal 2 do *timer* A (*flag* TACCR2:CCIFG) (símbolo **DMA0TSEL_1**, **DMA1TSEL_1** ou **DMA2TSEL_1**);

0010 - canal 2 do *timer* B (*flag* TBCCR2:CCIFG) (símbolo **DMA0TSEL_2**, **DMA1TSEL_2** ou **DMA2TSEL_2**);

0011 - recepção de dados na USART0 (*flag* URXIFG0) (símbolo **DMA0TSEL_3**, **DMA1TSEL_3** ou **DMA2TSEL_3**);

0100 - transmissão de dados completada na USART0 (*flag* UTXIFG0) (símbolo **DMA0TSEL_4**, **DMA1TSEL_4** ou **DMA2TSEL_4**);

0101 - DAC pronto para nova conversão (*flag* DAC12IFG) (símbolo **DMA0TSEL_5**, **DMA1TSEL_5** ou **DMA2TSEL_5**);

0110 - conversão completada no ADC (*flag* ADC12IFG) (símbolo **DMA0TSEL_6**, **DMA1TSEL_6** ou **DMA2TSEL_6**);

0111 - canal 0 do *timer* A (*flag* TACCR0:CCIFG);

1000 - canal 0 do *timer* B (*flag* TBCCR0:CCIFG);

1001 - recepção de dados na USART1 (*flag* URXIFG1);

1010 - transmissão de dados completada na USART1 (*flag* UTXIFG1);

1011 - multiplicador pronto;

1100 - nenhuma ação;

1101 - nenhuma ação;

1110 - disparo do outro canal de DMA: o canal 0 dispara o 1, o canal 1 dispara o 2 e o canal 2 dispara o 0 (símbolo **DMA0TSEL_14**, **DMA1TSEL_14** ou **DMA2TSEL_14**);

1111 - disparo externo pelo sinal DMAE0 (símbolo **DMA0TSEL_15**, **DMA1TSEL_15** ou **DMA2TSEL_15**).

5.19.8.2. DMACTL1

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0124	DMACTL1	Leitura	0	0	0	0	0	0	0	0
		Escrita	-	-	-	-	-	-	-	-
		Reset	0	0	0	0	0	0	0	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	0	0	0	0	0	DMA	ROUND	ENNMI
		Escrita	-	-	-	-	-	ONFETCH	ROBIN	
		Reset	0	0	0	0	0	0	0	0

DMAONFETCH - configura o DMA para ocorrer no ciclo de busca de instrução da CPU:

0 - a transferência DMA ocorre imediatamente;

1 - a transferência DMA ocorrerá na próxima busca de instrução depois do disparo (símbolo **DMAONFECTH**).

ROUND ROBIN - altera o esquema de prioridade do DMA:

0 - a prioridade do DMA é DMA0 - DMA1 - DMA2;

1 - a prioridade do DMA varia com cada transferência (símbolo **ROUNDROBIN**).

ENNMI - permite que uma interrupção NMI interrompa uma transferência do DMA:

0 - uma interrupção NMI não interrompe uma transferência DMA;

1 - uma interrupção NMI interrompe uma transferência DMA (símbolo ENNMI).

5.19.8.3. DMAxCTL

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
		Leitura	Reservado	DMADTx			DMADSTINCRx		DMASRCINCRx	
		Escrita								
0x01E0	DMA0CTL	*Reset*	0	0	0	0	0	0	0	0
0x01E8	DMA1CTL		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x01F0	DMA2CTL	Leitura	DMA	DMA	DMA	DMAEN	DMAIFG	DMAIE	DMA	DMAREQ
		Escrita	DST*BYTE*	SRC*BYTE*	LEVEL				ABORT	
		Reset	0	0	0	0	0	0	0	0

DMADTx - modo de transferência do DMA:

000 - transferência simples (símbolo **DMADT_0**);

001 - transferência de bloco (símbolo **DMADT_1**);

010 - transferência de bloco em rajadas (símbolo **DMADT_2**);

011 - transferência de bloco em rajadas (símbolo **DMADT_3**);

100 - transferência simples repetitiva (símbolo **DMADT_4**);

101 - transferência de bloco repetitiva (símbolo **DMADT_5**);

110 - transferência de bloco repetitiva em rajadas (símbolo **DMADT_6**);

111 - transferência de bloco repetitiva em rajadas (símbolo **DMADT_7**).

DMADSTINCRx - incremento do endereço de destino. A cada nova transferência o endereço de destino é automaticamente acrescido de um no caso de transferências de *byte* ou de dois no caso de transferências de *word*. O conteúdo do registrador DMAxDA não é alterado, apenas uma cópia dele, que é utilizada para o efetivo endereçamento:

00 - o endereço de destino não é alterado (símbolo **DMADSTINCR_0**);

01 - o endereço de destino não é alterado (símbolo **DMADSTINCR_1**);

10 - o endereço de destino é decrementado (símbolo **DMADSTINCR_2**);

11 - o endereço de destino é incrementado (símbolo **DMADSTINCR_3**).

DMASRCINCRx - incremento do endereço fonte. A cada nova transferência o endereço fonte é automaticamente acrescido de um no caso de transferências de *byte* ou de dois no caso de transferências de *word*. O conteúdo do registrador DMAxSA não é alterado, apenas uma cópia dele, que é utilizada para o efetivo endereçamento:

00 - o endereço fonte não é alterado (símbolo **DMASRCINCR_0**);

01 - o endereço fonte não é alterado (símbolo **DMASRCINCR_1**);

10 - o endereço fonte é decrementado (símbolo **DMASRCINCR_2**);

11 - o endereço fonte é incrementado (símbolo **DMASRCINCR_3**).

DMADSTBYTE - seleciona se o destino é um *byte* ou *word*:

0 - *word*;

1 - *byte* (símbolo **DMADSTBYTE**).

DMASRCBYTE - seleciona se a fonte é um *byte* ou *word*:

0 - *word*;

1 - *byte* (símbolo **DMASRCBYTE**).

DMALEVEL - seleciona se o disparo do DMA será por nível lógico ou por borda (transição do sinal):

0 - sensível à borda de subida;

1 - sensível ao nível lógico alto ("1") (símbolo **DMALEVEL**).

DMAEN - habilitação do DMA:

0 - DMA desabilitado;

1 - DMA habilitado (símbolo **DMAEN**).

DMAIFG - sinalizador de interrupção do DMA:

0 - nenhuma interrupção pendente;

1 - uma transferência DMA foi completada (símbolo **DMAIFG**).

DMAIE - habilitação de interrupção do DMA:

0 - interrupção desabilitada;

1 - interrupção habilitada (símbolo **DMAIE**).

DMAABORT - indica se uma transferência DMA foi interrompida por uma interrupção NMI:

0 - nenhuma transferência foi interrompida;

1 - uma transferência DMA foi interrompida por uma interrupção NMI (símbolo **DMAABORT**).

DMAREQ - requisição de DMA por *software*, o *bit* é apagado automaticamente:

0 - não iniciar uma transferência DMA;

1 - iniciar uma transferência DMA (símbolo **DMAREQ**).

5.19.8.4. DMAxSA

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x01E2	DMA0SA	Leitura	DMAxSA				
0x01EA	DMA1SA	Escrita					
0x01F2	DMA2SA	*Reset*	?	?	...	?	?

DMAxSA - endereço fonte da transferência DMA. Esse registrador aponta para o endereço do elemento ou dos elementos a serem transferidos pelo DMA. O conteúdo desse registrador não é alterado nas transferências de bloco.

5.19.8.5. DMAxDA

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x01E4	DMA0DA	Leitura	DMAxDA				
0x01EC	DMA1DA	Escrita					
0x01F4	DMA2DA	*Reset*	?	?	...	?	?

DMAxDA - endereço de destino da transferência DMA. O conteúdo desse registrador não é alterado nas transferências de bloco.

5.19.8.6. DMAxSZ

Endereço	Nome		*BIT* 15	*BIT* 14	...	*BIT* 1	*BIT* 0
0x01E6	DMA0SZ	Leitura	DMAxSZ				
0x01EE	DMA1SZ	Escrita					
0x01F6	DMA2SZ	*Reset*	?	?	...	?	?

DMAxSZ - tamanho do bloco a ser transferido pelo DMA. O seu conteúdo é automaticamente decrementado a cada transferência até que atinja o valor 0, quando o seu valor prévio é recarregado:

0000 - transferência desabilitada;

0001 - transferência de um *byte* ou *word*;

0010 - transferência de dois *bytes* ou *words*;

...

1111 - transferência de 65535 *bytes* ou *words*.

5.19.9. Exemplos de Utilização

O primeiro exemplo de utilização do DMA é uma função de transferência de um bloco da memória utilizando apenas o controlador de DMA. O programa foi testado em um dispositivo MSP430F169.

```
//*******************************************************************
// Exemplo de utilização do DMA
//*******************************************************************
// Autor: Fábio Pereira
```

```
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*********************************************************************
// Este programa utiliza o canal 0 do controlador de DMA para fazer
// a transferência de um bloco de memória de um endereço para outro
// O conteúdo da matriz temp é copiado para a matriz temp2
//*********************************************************************
#include <io430x16x.h>

unsigned char temp [10] =
   {
            1,2,3,4,5,
            6,7,8,9,0
   };
unsigned char temp2 [10] =
   {
            0,0,0,0,0,
            0,0,0,0,0
   };

//*********************************************************************
// copia_bloco_dma
//*********************************************************************
// Entrada:         char *   end_origem - o endereço inicial do bloco de
//                           origem
//                  char * end_destino - o endereço inicial do bloco de
//                           destino
//                  unsigned int tamanho - o número de bytes a ser trans-
//                           ferido
// Saída:           void
//*********************************************************************
// A função copia_bloco_dma utiliza o canal 0 do controlador de DMA
// para copia um bloco de memória localizado a partir do endereço
// end_origem para o bloco localizado a partir de end_destino
// A quantidade de bytes a serem transferidos é passada através do
// parâmetro tamanho
//*********************************************************************
void copia_bloco_dma(char * end_origem, char * end_destino, unsigned int tamanho)
{
   // configura o controlador de DMA
   DMACTL0 = 0;
   // configura o canal 0 do controlador
   DMA0CTL = DMADT_1 + DMADSTINCR_3 + DMASRCINCR_3 + DMADSTBYTE + DMASRCBYTE + DMAEN;
   // configura o endereço do bloco de origem
   DMA0SA = (int)end_origem;
   // configura o endereço do bloco de destino
   DMA0DA = (int)end_destino;
   // configura o tamanho da transferência em bytes
   DMA0SZ = tamanho;
   // inicia a transferência
   DMA0CTL_bit.DMAREQ = 1;
}

void main( void )
{
   // copia a matriz temp para a matriz temp2
   copia_bloco_dma(temp, temp2, 10);
   while (1);
}
```

Exemplo 5-17

O próximo exemplo demonstra a utilização do DMA para automatizar a tarefa de multiplicação de uma matriz de 15 elementos por um coeficiente fixo. Toda a operação é realizada automaticamente sem a intervenção da CPU.

```
//*********************************************************************
// Exemplo de multiplicação por hardware utilizando DMA
//*********************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*********************************************************************
```

```
// Este programa multiplica os elementos de uma matriz pelo valor
// 5 e armazena novamente na matriz. A operação é feita utilizando
// o multiplicador por hardware e os canais 0 e 1 do DMA. Este exem-
// plo foi testado em um MSP430F169 utilizando a placa Microlab X1
//*******************************************************************

#include <io430x16x.h>

volatile int matriz[15] =
   {
            200, -50, 920, 333, 477,
            1500, 25, -670, -344, 3517,
            9999, 2000, 555, 190, -2345
   };

void main( void )
{
   // carrega a constante no registrador MPYS
   // isto seleciona uma multiplicação sinalizada
   MPYS = 5;
   // configura os canais 0 e 1 para disparo pelo
   // multiplicador. Como o canal 0 possui prioridade
   // maior que o 1, o mesmo é utilizado para transferir
   // o resultado da multiplicação para a matriz, en-
   // quanto que o canal 1 é utilizado para transferir
   // o próximo elemento da matriz para o multiplicador
   DMACTL0 = 0xBB;
   // configura o canal 0 para incrementar o endereço
   // de destino a cada transferência. O endereço fonte
   // é fixo.
   DMA0CTL =DMADSTINCR_3 + DMAEN;
   // carrega o endereço do registrador do resultado
   // da multiplicação no registrador de endereço fonte
   // do canal 0 do DMA
   DMA0SA = (int) (&RESLO);
   // carrega o endereço do primeiro elemento da matriz
   // no registrador de endereço de destino do canal 0
   // do DMA
   DMA0DA = (int) matriz;
   // o número de transferências a serem realizadas
   // pelo canal 0 é igual a 15
   DMA0SZ = 15;
   // configura o canal 1 para incrementar o endereço
   // fonte a cada transferência. O endereço de destino
   // é mantido fixo.
   DMA1CTL = DMASRCINCR_3 + DMAEN;
   // o endereço do segundo elemento da matriz é carregado
   // no registrador de endereço fonte do canal 1
   DMA1SA = ((int) matriz)+2;
   // o endereço do registrador do operando 2 do multi-
   // plicador é carregado no registrador de endereço de
   // destino no canal 1
   DMA1DA = (int) (&OP2);
   // o número de transferências do canal 1 é igual a 14
   // pois a primeira carga do OP2 é feita manualmente
   DMA1SZ = 14;
   // carrega o registrador OP2 com o primeiro elemento
   // da matriz. Isto inicia a operação do multiplicador.
   // Ao término da primeira multiplicação, o DMA0
   // transfere o resultado para a matriz e o DMA1 carre-
   // ga um novo valor da matriz no multiplicador. Este
   // processo segue até que a quantidade de transfe-
   // rências carregada nos registradores DMAxSZ tenham
   // sido realizadas
   OP2 = matriz[0];
   while (1);
}
```

Exemplo 5-18

5.20. Supervisor de Tensão

O módulo supervisor de tensão (SVS) consiste basicamente em um circuito de monitoração da tensão de alimentação do *chip* (a tensão de alimentação analógica AV_{CC} ou uma tensão externa pela entrada SVSIN), capaz de gerar uma interrupção ou inclusive um *reset* do sistema no caso de a tensão cair abaixo de um valor programado. O SVS possui um desenho de baixa potência, com um consumo típico, quando ativo, da ordem de 10μA.

O seu princípio de funcionamento é muito simples: o coração do circuito é um comparador analógico. A sua entrada positiva está conectada a uma referência de tensão de 1,25 Volts. A entrada negativa está conectada, por um multiplexador analógico, à fonte de tensão a ser monitorada (AV_{CC} ou SVSIN).

A função do multiplexador é selecionar um entre vários divisores da tensão monitorada, permitindo que se selecionem níveis de disparo do comparador. A seleção da entrada do multiplexador é feita pelos *bits* VLDx (registrador SVSCTL).

A saída do comparador pode ser monitorada pelo *bit* SVSOP (registrador SVSCTL) e uma vez ativada, provoca a ativação do *flag* SVSFG, que permanece ativado até que seja apagado pelo *software* do usuário, sinalizando a ocorrência de uma queda da tensão monitorada.

Caso o *bit* PORON esteja setado, a ativação do SVSFG provoca um *reset* do *chip*.

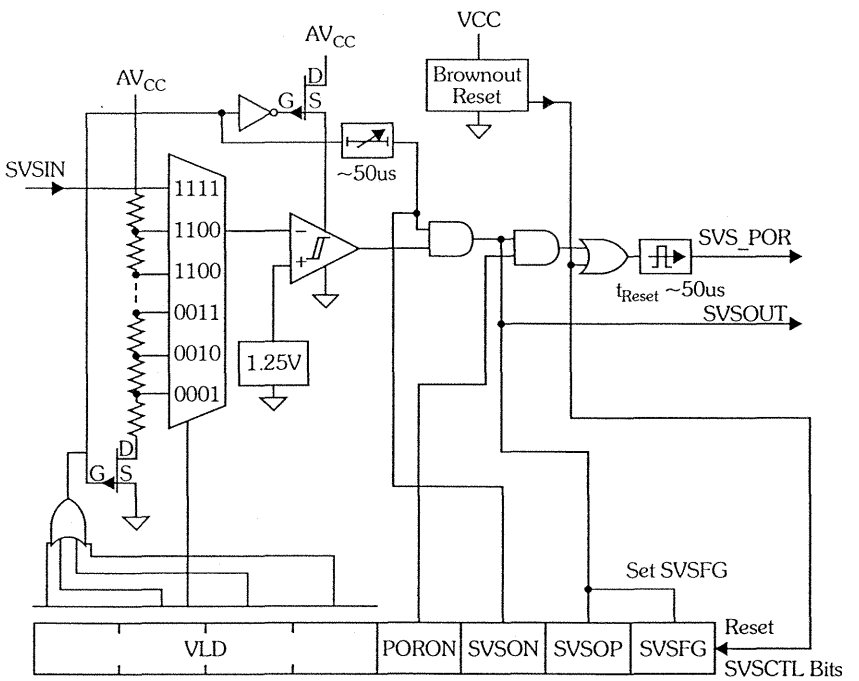


Figura 5-51

O SVS pode ser encontrado nos modelos 15x, 16x e todos os *chips* da família 4xx.

5.20.1. Conexões do SVS

O controlador de DMA pode utilizar os seguintes pinos:

Sinal	Pino
SVSIN	P6.7
SVSOUT	P5.7 (família 1xx) P1.3 (família 4xx)

Tabela 5-55

5.20.2. Registradores do Supervisor de Tensão

O módulo supervisor de tensão utiliza apenas o registrador SVSCTL para o controle do seu funcionamento.

5.20.2.1. SVSCTL

Endereço	Nome		*BIT* 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
0x0055 0x0056	SVSCTL	Leitura	VLDx				PORON	SVSON	SVSOP	SVSFG
		Escrita						-	-	
		Reset	0	0	0	0	0	?	?	0

* O endereço do registrador SVSCTL é 0x0055 nos *chips* das famílias 1xx e 2xx e 0x0056 nos *chips* da família 4xx.

VLDx - nível de tensão de disparo:

0000 - SVS desligado (símbolo **VLDOFF**);
0001 - 1,9V;
0010 - 2,1V;
0011 - 2,2V;
0100 - 2,3V;
0101 - 2,4V;
0110 - 2,5V;
0111 - 2,65V;
1000 - 2,8V;
1001 - 2,9V;
1010 - 3,05V;
1011 - 3,2V;
1100 - 3,35V;
1101 - 3,5V;
1110 - 3,7V;
1111 - compara a tensão externa lida em SVSIN com 1,25V;

*** Nos modelos 412 e 413, somente a tensão de 1,9 Volts está disponível.**

PORON - habilita o *flag* SVSFG a causar um *reset* POR quando ele for setado:

0 - o *flag* SVSFG quando setado não causa um POR;

1 - o *flag* SVSFG quando setado causa um POR (símbolo **PORON**).

SVSON - permite **ler** o estado de funcionamento do SVS:

0 - SVS desligado;

1 - SVS ligado (símbolo **SVSON**).

SVSOP - esse *bit* reflete o estado do comparador de saída do SVS:

0 - saída do comparador ativa (nível "1");

1 - saída do comparador inativa (nível "0") (símbolo **SVSOP**).

SVSFG - sinalizador de baixa tensão:

0 - a tensão de alimentação está acima do limite mínimo programado;

1 - a tensão de alimentação está (ou esteve) abaixo do limite mínimo programado (símbolo **SVSFG**).

5.21. Controlador da Memória *FLASH*

Todos os modelos de MSP430 abordados no livro utilizam memória *FLASH* para o armazenamento do programa.

O controlador de memória *FLASH* é um circuito implementado nesses *chips* de forma a controlar o processo de apagamento e programação dessa memória.

Pode-se programar desde um único *bit* até vários *bytes* ou *words*, enquanto o apagamento se dá por segmentos de memória, como veremos mais adiante.

O controlador inclui ainda um circuito de temporização, encarregado de gerar as temporizações necessárias ao processo de programação e apagamento da memória, além do *clock* necessário ao circuito gerador de tensão de programação. O *clock* desse gerador pode ser selecionado para ter início de um dos três sinais de *clock* internos do MCU (MCLK, SMCLK ou ACLK). O sinal pode ainda ser dividido por um fator entre 1 e 64, antes de originar o *clock* do controlador da *FLASH* (sinal FTG).

Outra característica interessante é que todos os registradores do controlador da memória *FLASH* são protegidos com senhas de escrita, o que impede que escritas espúrias neles, possam iniciar operações de apagamento/programação indesejadas.

O valor escrito em tais registradores deve sempre possuir o *byte* superior igual a 0xA5 (que é a senha de escrita). Uma operação de escrita utilizando um valor diferente deste faz com que o *flag* KEYV (registrador FCTL3) seja setado, provocando um *reset* da CPU.

Observe ainda que o consumo geral do *chip* sobe para cerca de 3mA durante uma operação de apagamento ou programação.

5.21.1. Apagamento da Memória

Nas operações de apagamento da memória *FLASH*, ela é dividida em blocos chamados de segmentos. Um segmento é a menor porção de memória que pode ser apagada.

Existem dois tipos de segmento: os da memória principal, com um tamanho de 512 *bytes*, e os da memória de informação (*information memory*), com um tamanho de 128 *bytes* (64 *bytes* na família 2xx). A tabela 5-56 apresenta os limites de endereçamento dos primeiros segmentos da memória *FLASH*. Repare que a partir do nono segmento (o de número 8) temos apenas a mudança do primeiro dígito do endereço hexadecimal.

Endereço	Segmento
0xFFFF	0
0xFE00	
0xFDFF	1
0xFC00	
0xFBFF	2
0xFA00	
0xF9FF	3
0xF800	
0xF7FF	4
0xF600	
0xF5FF	5
0xF400	

Endereço	Segmento
0xF3FF	6
0xF200	
0xF1FF	7
0xF000	
0xEFFF	8
0xEE00	
...	
0x11FF	119
0x1100	

Tabela 5-56

Nos *chips* com 55 ou 60Kb de memória *FLASH*, o último segmento (endereços 0x2500 a 0x25FF para o MSP430F1612 e 0x1100 a 0x11FF para os MSP430F149, 1491, 169, FG439 e F449) possui um tamanho de apenas 256 *bytes*.

A memória de informação é uma porção especial de memória *FLASH* localizada entre os endereços 0x1000 e 0x10FF. Essa porção de memória é utilizada pelo fabricante para armazenar valores de calibração ou configuração em alguns modelos de *chips*. No entanto, na maioria dos casos, essa região pode ser utilizada para o armazenamento de informações diversas (o fabricante recomenda o seu apagamento antes da utilização).

Existem normalmente dois segmentos de memória de informação (A e B, cada um com 128 *bytes*), com exceção dos modelos 1.101, que possuem somente um (o segmento A). A tabela 5-57 apresenta os endereços válidos para os segmentos da memória de informação.

Na família 2xx, cada segmento possui 64 *bytes* num total de quatro segmentos (A, B, C e D). A tabela 5-58 apresenta os endereços válidos para os segmentos da memória de informação na família 2xx.

Endereço	Segmento
0x10FF	A
0x1080	
0x107F	B
0x1000	

Tabela 5-57

Endereço	Segmento
0x10FF 0x10C0	A
0x10BF 0x1080	B
0x107F 0x1040	C
0x103F 0x1000	D

Tabela 5-58

Existem três modalidades de apagamento disponíveis: apagamento de um segmento, apagamento de todos os segmentos da memória principal e apagamento de todos os segmentos da memória principal mais os segmentos da memória de informação. A seleção do modo de apagamento é feita pelos *bits* MERAS (***Mass ERASe*** - apagamento em massa) e ERASE (apagamento), localizados no registrador FCTL1.

MERAS	ERASE	Modo de apagamento
0	0	Desligado
0	1	Apagamento de um segmento apenas
1	0	Apagamento de toda a *FLASH*
1	1	Apagamento da *FLASH* e da memória de informação

Tabela 5-59

Além desses *bits*, é necessário configurar o gerador de *clock* do controlador da memória *FLASH* e apagar o *bit* de proteção contra gravação (FCTL3:LOCK).

Também é importante desabilitar o *watchdog* e as interrupções (tanto desligando o GIE como também os controles das interrupções NMI) antes do início da operação de apagamento, de forma a evitar perturbações ao processo de apagamento.

Feito isso, basta realizar uma operação de escrita em qualquer endereço dentro daquele segmento ou região selecionada, para iniciar o processo de apagamento. Uma escrita em um endereço não pertencente ao segmento/região é descartada e não produz nenhum efeito.

Após o início de uma operação de apagamento, o *flag* BUSY (registrador FCTL3) é setado, indicando que o controlador está ocupado. Os *bits* BUSY, MERAS e ERASE são apagados automaticamente pelo controlador após o término da operação de apagamento.

A duração de um ciclo de apagamento de um segmento é de 4819 ciclos de *clock* FTG e para o apagamento geral da *FLASH* é de 5.297 ciclos FTG.

Observe que uma operação de apagamento pode ser iniciada tanto por um código rodando na memória *FLASH* como por um código rodando na memória RAM.

No primeiro caso a CPU é mantida automaticamente num *loop* utilizando a instrução JMP PC até o término da operação de apagamento. Após isso, a execução do programa continua da instrução seguinte a que iniciou o apagamento.

No segundo caso, a execução do programa segue a partir da memória RAM. No entanto, o programa não pode fazer nenhum acesso à memória *FLASH* antes que o *flag* BUSY (registrador FCTL3) seja apagado; caso contrário, ocorre uma violação de acesso à memória *FLASH* e o *flag* ACCVIFG será setado.

Encerrado o ciclo de apagamento, as interrupções e o *watchdog* podem ser novamente habilitados e o programa pode acessar a memória *FLASH* normalmente.

Na família 2xx, é possível manter as interrupções habilitadas, sendo permitido inclusive que uma interrupção encerre prematuramente uma operação de apagamento (caso o *bit* FCTL1:EEIEX esteja setado), ou interrompa a operação (caso o *bit* FCTL1:EEI esteja setado). Neste último caso, após o tratamento da interrupção, a operação de apagamento é retomada do ponto em que havia sido interrompida.

5.21.2. Programando a *FLASH*

Como já foi dito, é possível endereçar e programar desde um único *bit* até vários *bytes* e *words* da memória *FLASH*. Adicionalmente, também é possível programar blocos de 64 *bytes* de memória, o que diminui o tempo de programação substancialmente.

A seleção do modo de programação é controlada basicamente pelos *bits* WRT e BLKWRT (localizados no registrador FCTL1). O *bit* WRT seleciona o modo de programação (WRT = 1), enquanto o *bit* BLKWRT seleciona o modo de programação em blocos (BLKWRT = 1), ou *bit/byte/word* (BLKWRT = 0). Além desses *bits*, é necessário apagar o *bit* LOCK (registrador FCTL3) antes do início da operação.

A programação de uma ou mais posições de memória pode ser iniciada por qualquer instrução cujo destino seja o endereço a ser programado.

Antes de iniciar a programação, é importante desabilitar o *watchdog* e as interrupções (tanto as mascaráveis como as não-mascaráveis), de forma a evitar problemas de acesso a memória durante o processo de programação.

O tempo necessário para programar uma posição de memória (*bit/byte/word*) é de 35 ciclos FTG. Utilizando o modo de programação em blocos, o gerador de tensão de programação permanece ligado durante todo o processo e o tempo de programação reduz-se a 30 ciclos para o primeiro *byte* e 21 ciclos para os *bytes* restantes (até o máximo de 63). Um tempo adicional de seis ciclos é necessário para a desativação do gerador de tensão.

Durante o processo de programação, o *flag* BUSY permanece setado, indicando que o controlador da memória está ocupado. O acesso a memória *FLASH* somente é permitido após esse *flag* ser apagado (FCTL3:BUSY = 0), o que ocorre automaticamente ao término do ciclo de

programação. Caso o *software* realize um acesso à memória *FLASH* enquanto o *flag* BUSY = 1, ocorrerá uma violação de acesso à memória e o *flag* ACCVIFG será setado.

A programação da *FLASH* pode ser iniciada tanto de um código contido na própria memória, como de um código em execução na memória RAM.

No primeiro caso a CPU é mantida automaticamente num *loop* utilizando a instrução JMP PC até o término da operação de programação. Após a sua conclusão, a execução do programa continua da instrução seguinte a que iniciou a operação. Neste caso, somente é possível a programação de *bits, bytes* ou *words* e não de um bloco de memória.

No segundo caso, a execução do programa segue a partir da memória RAM. No entanto, o programa não pode fazer nenhum acesso à memória *FLASH* antes que o *flag* FCTL3:BUSY seja apagado; caso contrário, ocorre uma violação de acesso à memória *FLASH* e o *flag* ACCVIFG será setado.

Encerrado o ciclo de programação, as interrupções e o *watchdog* podem ser novamente habilitados e o programa pode acessar a memória *FLASH* normalmente.

5.21.2.1. Programação de Blocos

A programação de blocos de memória somente pode ser iniciada por um programas em execução a partir da RAM.

Para a programação de um bloco de memória o *software* deve inicialmente desativar o *watchdog* e as interrupções (tanto as mascaráveis como as não-mascaráveis).

Em seguida, o *software* deve aguardar que o *flag* BUSY seja apagado (caso uma outra operação de programação ou apagamento ainda esteja em andamento). Também o *flag* WAIT deve ser verificado. O processo de programação somente pode ser iniciado com esse *flag* em nível lógico "0".

Após estas condições estarem satisfeitas, os *bits* BLKWRT e WRT devem ser setados e o *bit* LOCK apagado. Feito isso, o processo de escrita na memória *FLASH* pode ter início.

A cada *byte* ou *word* escrita o *flag* WAIT é automaticamente apagado e um novo *byte/word* somente pode ser escrito após ele retornar ao nível "1" (o que ocorre cerca de 21 ciclos FTG após a operação de escrita).

Após o último *byte* ou *word* ser escrito (é possível escrever blocos menores que 64 *bytes*), o *bit* BLKWRT deve ser apagado. Isso encerra a programação do bloco corrente. Um tempo adicional de seis ciclos FTG é necessário para que a tensão de programação seja removida.

Durante todo o processo de programação, o *flag* FCTL3:BUSY permanece setado, somente sendo apagado após o término da operação. Somente após o término completo do ciclo de programação de um bloco é que a programação de outro pode ser iniciada.

Nos intervalos entre a escrita de cada bloco (caso mais de um esteja sendo programado), pode ser interessante habilitar as interrupções, de forma que o programa possa atender às que porventura estejam pendentes. No entanto, antes do início da escrita do outro bloco, elas deverão ser novamente desativadas.

5.21.3. Encerrando Prematuramente uma Operação

É possível abortar uma operação de apagamento ou programação da memória *FLASH* antes que encerre naturalmente.

O *bit* FCTL3:EMEX (***EM**ergency **EX**it* - saída de emergência) setado provoca o encerramento imediato da operação de apagamento/programação, fazendo com que o controlador da *FLASH* seja desativado e a memória retorne ao seu modo normal de leitura. Todos os *bits* do registrador FCTL1 são apagados.

Obviamente, o resultado da operação que se encontrava em andamento é indefinido, o que implica que ela deve ser obrigatoriamente repetida.

5.21.4. Interrupções do Controlador da *FLASH*

Além da violação de senha (*flag* KEYV), que é disparada após uma operação de escrita em um dos registradores do controlador da *FLASH* utilizando uma senha incorreta, o controlador utiliza ainda a interrupção NMI de violação de acesso à memória *FLASH* (*flag* ACCVIFG).

Esse *flag* é setado nas seguintes condições:

1. Uma operação de leitura/escrita da *FLASH* ocorra enquanto o *flag* BUSY = 1.
2. Numa operação de escrita no registrador FCTL1 durante um ciclo de apagamento ou de programação de *bit/byte/word*.
3. Numa operação de escrita no registrador FCTL1 durante um ciclo de programação de bloco enquanto o *flag* WAIT = 0. Neste caso, somente é possível escrever no FCTL1 quando WAIT = 1.
4. Numa operação de escrita no registrador FCTL2.

Com o *flag* FCTL3:ACCVIFG setado e caso essa fonte de interrupção NMI esteja habilitada (IE1:ACCVIE = 1), uma interrupção NMI será disparada, provocando o desvio do programa para o endereço indicado pelo vetor 14 (veja a tabela 5-1).

5.21.5. Registradores do Controlador de Memória *FLASH*

O controlador de memória *FLASH* utiliza quatro registradores para o seu funcionamento:

- **FCTL1, FCTL2 e FCTL3** - para o controle do módulo.
- **IE1** - para o controle de interrupção ACCVIE.

5.21.5.1. FCTL1

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0128	FCTL1	Leitura	FRKEY - lida sempre como 0x96							
		Escrita	FWKEY - deve sempre ser escrito 0xA5							
		Reset	1	0	0	1	0	1	1	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	BLKWRT	WRT	Reservado	EEIEX*	EEI*	MERAS	ERASE	Reservado
		Escrita			-					-
		Reset	0	0	0	0	0	0	0	0

FRKEY - valor-padrão para a leitura do *byte* superior do registrador. É sempre igual a 0x96 (símbolo **FRKEY**).

FWKEY - senha para escrita no registrador. O valor a ser escrito deve ser sempre igual a 0xA5, sob pena de o controlador gerar um *reset* (símbolo **FWKEY**).

BLKWRT - modo de programação em blocos. O *bit* WRT deve também ser setado para a programação de blocos. Esse *bit* é automaticamente apagado quando o *bit* EMEX é setado:

0 - programação em blocos desativada;

1 - programação em blocos ativada (símbolo **BLKWRT**).

WRT - ativa o modo de escrita:

0 - modo de escrita desativado;

1 - modo de escrita ativado (símbolo **WRT**).

EEIEX - interrupção de saída de emergência (somente na família 2xx):

0 - uma interrupção não causa uma saída de emergência;

1 - uma interrupção causa uma saída de emergência (símbolo **EEIEX**).

EEI - habilitação de interrupção durante o apagamento (somente na família 2xx):

0 - interrupções desabilitadas;

1 - interrupções habilitadas (símbolo **EEI**).

MERAS - apagamento em massa (símbolo **MERAS**).

ERASE - (símbolo **ERASE**) os sinais MERAS e ERASE selecionam uma das modalidades de apagamento conforme a tabela seguinte:

MERAS	ERASE	Modo de apagamento
0	0	Desligado
0	1	Apagamento de um segmento apenas
1	0	Apagamento de toda a *FLASH*
1	1	Apagamento da *FLASH* e da memória de informação

5.21.5.2. FCTL2

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x012A	FCTL2	Leitura	FRKEY - lida sempre como 0x96							
		Escrita	FWKEY - deve sempre ser escrito 0xA5							
		Reset	1	0	0	1	0	1	1	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	FSSELx		FNx					
		Escrita								
		Reset	0	1	0	0	0	0	1	0

FRKEY - valor-padrão para a leitura do *byte* superior do registrador. É sempre igual a 0x96 (símbolo **FRKEY**).

FWKEY - senha para escrita no registrador. O valor a ser escrito deve ser sempre igual a 0xA5, sob pena de o controlador gerar um *reset* (símbolo **FWKEY**).

FSSELx - seleção da fonte de *clock* do controlador da memória *FLASH*:

00 - *clock* auxiliar (ACLK) (símbolo **FSSEL_0**);

01 - *clock* principal (MCLK) (símbolo **FSSEL_1**);

10 - *clock* secundário (SMCLK) (símbolo **FSSEL_2**);

11 - *clock* secundário (SMCLK) (símbolo **FSSEL_3**).

FNx - fator de divisão do *clock* do controlador da memória *FLASH*, o *clock* é dividido por um fator igual a FNx + 1 (símbolos **FN0, FN1, FN2, FN3, FN4** e **FN5**).

5.21.5.3. FCTL3

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x012C	FCTL3	Leitura	FRKEY - lida sempre como 0x96							
		Escrita	FWKEY - deve sempre ser escrito 0xA5							
		Reset	1	0	0	1	0	1	1	0
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura / Escrita	FAIL*	LOCKA*	EMEX	LOCK	WAIT	ACCVIFG	KEYV	BUSY
		Reset	0	1	0	1	1	0	0	0

* Somente na família 2xx.

FRKEY - valor-padrão para a leitura do *byte* superior do registrador. É sempre igual a 0x96 (símbolo **FRKEY**).

FWKEY - senha para escrita no registrador. O valor a ser escrito deve ser sempre igual a 0xA5 sob pena de o controlador gerar um *reset* (símbolo **FWKEY**).

FAIL - indicação de falha na operação. É setado quando ocorre uma falha no *clock* do controlador da *FLASH* ou se uma operação é abortada por uma interrupção (quando FCTL1:EEIEX = 1) (somente disponível na família 2xx):

0 - Nenhuma falha;

1 - Falha na operação (símbolo **FAIL**).

LOCKA - Proteção contra apagamento da memória de informação e contra apagamento e escrita do segmento A (somente na família 2xx):

0 - funcionamento normal;

1 - operação abortada (símbolo **LOCKA**).

EMEX - saída de emergência. Permite abortar uma operação de escrita ou apagamento que esteja em andamento. Os *bits* do registrador FCTL1 são apagados:

0 - funcionamento normal;

1 - operação abortada (símbolo **EMEX**).

LOCK - proteção da memória. Esse *bit,* quando setado, impede a realização de operações de escrita ou apagamento:

0 - operações de escrita/apagamento liberadas;

1 - operações de escrita/apagamento inibidas, memória protegida (símbolo **LOCK**).

WAIT - indica que uma operação de escrita ou apagamento está em andamento:

0 - operação em andamento;

1 - o controlador está pronto para uma nova operação (símbolo **WAIT**).

ACCVIFG - sinalizador da interrupção de violação de acesso:

0 - nenhuma interrupção pendente;

1 - houve uma violação de acesso (ocorreu uma escrita no registrador FCTL1 quando WAIT estava em 0, ou um acesso à *FLASH* quando BUSY estava em 1) (símbolo **ACCVIFG**).

KEYV - sinalizador da interrupção de violação de segurança da *FLASH*:

0 - nenhuma interrupção pendente;

1 - houve uma violação de segurança (ocorreu uma escrita em um dos registradores do controlador da *FLASH* com uma senha diferente de 0xA5) (símbolo **KEYV**).

Lembre-se de que essa interrupção não é mascarável !

BUSY- estado do circuito de temporização do controlador da *FLASH*:
0 - livre;
1 - ocupado (operação de programação em andamento) (símbolo **BUSY**).

5.21.5.4. IE1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0000	IE1	Leitura / Escrita	UTXIE0	URXIE0	ACCVIE	NMIIE	-	-	OFIE	WDTIE
		Reset	0	0	0	0	0	0	0	0

ACCVIE - habilitação da interrupção por violação de acesso à memória *FLASH*.

5.21.6. Exemplos de Utilização

A seguir temos algumas funções que podem ser utilizadas para escrever e ler a memória *FLASH* (no caso a memória de informação) nos *chips* das famílias 1xx e 4xx. O procedimento para escrita em outras regiões da memória *FLASH* é basicamente o mesmo. Basta lembrar que o tamanho do segmento nessas regiões é maior que o da memória de informação.

```
//*************************************************************************
//* Função apaga_MI - apagamento da memória de informação
//*************************************************************************
//* Argumentos de entrada : nenhum
//* Retorno : nada
//*************************************************************************
//* Observações:
//* Apaga o conteúdo da memória de informação
//*************************************************************************
void apaga_MI(void)
{
   char *Flash_ptr;
   Flash_ptr = (char*) 0x1000;
   FCTL2 = FWKEY + FSSEL_0 + (clock / 400);
   FCTL1 = FWKEY + ERASE;
   FCTL3 = FWKEY;
   __disable_interrupt();
   *Flash_ptr = 0;
   __enable_interrupt();
   FCTL1 = FWKEY;
   FCTL3 = FWKEY + LOCK;
}

//*************************************************************************
//* Função escreve_MI - escrita na memória de informação
//*************************************************************************
//* Argumentos de entrada :
//* unsigned char endereco - endereço a ser escrito
//* unsigned int valor - dado a ser escrito na memória
//*************************************************************************
//* Retorno :
//* unsigned int - o dado que lido da posição programada
//*************************************************************************
```

```
//* Observações:
//* Esta função pode ser utilizada para programar uma posição de 16 bits
//* da memória de informação. Esta posição deve estar previamente apagada
//*************************************************************************
unsigned int escreve_MI(unsigned char endereco, unsigned int valor)
{
   int *Flash_ptr;
   __disable_interrupt();
   FCTL2 = FWKEY + FSSEL_0 + (clock / 400);
   FCTL3 = FWKEY;
   FCTL1 = FWKEY + WRT;
   Flash_ptr = (int*) 0x1000 + endereco;
   *Flash_ptr = valor;
   FCTL1 = FWKEY;
   FCTL3 = FWKEY + LOCK;
   __enable_interrupt();
   return (le_MI(endereco));
}

//*************************************************************************
//* Função le_MI - leitura da memória de informação
//*************************************************************************
//* Argumentos de entrada :
//* unsigned char endereco - o endereço de memória a ser lido
//*************************************************************************
//* Retorno :
//* unsigned int - o dado de 16 bits lido da memória
//*************************************************************************
//* Observações:
//* Nenhuma
//*************************************************************************
unsigned int le_MI(unsigned char endereco)
{
   int *Flash_ptr;
   Flash_ptr = (int*) 0x1000 + endereco;
   return (*Flash_ptr);
}
```

Exemplo 5-19

A utilização das funções é muito simples:

```
apaga_MI(); // apaga a memória de informação

// escreve o valor 5 na posição 0 da memória de informação
// a variável x recebe o valor lido da mesma posição
// neste caso, se x for diferente de 5, houve um erro de gravação
x = escreve_MI(0,5);

// também podemos chamar a função sem atribuir o seu resultado:
escreve_MI(0,5);

x = le_MI(2); // lê o endereço 2 da memória de informação
```

5.22. *Watchdog*

Todos os microcontroladores MSP430 incluem um temporizador cão de guarda (*watchdog*) projetado com a finalidade de "vigiar" a execução do *software*.

O *watchdog* consiste basicamente em um contador de 16 *bits* que, ao final do período de contagem (que pode ser selecionado pelo usuário), pode provocar um *reset* da CPU.

Opcionalmente, é possível programar o *watchdog* para apenas gerar uma interrupção e neste caso, ele passa a funcionar apenas como um temporizador.

Doravante, estudaremos o funcionamento do *watchdog* baseados nessas duas modalidades de operação. Antes, no entanto, vamos observar algumas características comuns às duas modalidades de funcionamento:

A base de operação do *watchdog* é um contador de 16 *bits* cuja fonte de *clock* pode ser selecionada entre duas opções: o sinal de *clock* secundário (SMCLK) ou o sinal de *clock* auxiliar (ACLK). A seleção é feita pelo *bit* WDTSSEL, localizado no registrador WDTCTL.

Quatro das saídas do contador podem ser utilizadas para provocar o *reset* do sistema ou interrupção. Cada uma delas corresponde ao sinal de *clock* do contador dividido por um fator (32.768, 8.192,512 ou 64). A seleção do fator de divisão é feita pelos *bits* WDTCTL:WDTISx.

Um outro detalhe importante diz respeito ao registrador de controle do *watchdog*: o WDTCTL é um registrador protegido por senha, isto é, a escrita nele é controlada pelo *hardware* do microcontrolador. Se durante a escrita o conteúdo do *byte* mais significativo não corresponder ao valor 0x5A, o conteúdo do restante do registrador não será alterado e ocorrerá um *reset* do sistema.

É possível desligar o módulo de forma a economizar energia, o que pode ser feito pelo *bit* WDTHOLD (registrador WDTCTL).

Observe que durante os modos de baixo consumo que desabilitem a fonte de *clock* do *watchdog*, ele permanecerá inativo até que a fonte de *clock* seja novamente ativada (com a saída do modo de baixo consumo).

Vejamos agora alguns detalhes sobre o funcionamento do *watchdog* em cada uma das suas modalidades de operação:

5.22.1. Modo *Watchdog*

Nesse modo, uma vez que o contador realize 32.768, 8.192, 512 ou 64 contagens (conforme a seleção dos *bits* WDTISx), será provocado um *reset* do sistema.

O programador deve providenciar o apagamento periódico da contagem do *watchdog*, de forma que ele nunca atinja a sua contagem final. Desta forma, caso a aplicação esteja funcionando corretamente, nenhum *reset* será provocado por ele.

Caso ocorra uma condição anômala qualquer que provoque o funcionamento incorreto do programa, muito provavelmente o *watchdog* não terá mais a sua contagem apagada, fazendo com que ele sete o *flag* WDTIFG (registrador IFG1) e gere um *reset* da CPU ao atingir a sua contagem final.

Nesse momento, deve ficar claro ao programador que a decisão sobre a utilização do *watchdog* nesse modo, bem como a posição do programa em que será realizado o apagamento da sua contagem, é de extrema importância para o correto funcionamento da aplicação. O uso incorreto do *watchdog* pode, muitas vezes, acabar gerando mais problemas do que soluções propriamente ditas.

O apagamento da contagem do *watchdog* pode ser realizado setando o *bit* WDTCNTCL no registrador WDTCTL.

Algumas dicas importantes sobre a utilização do *watchdog*:

- Utilize períodos de tempo compatíveis com a carga de processamento da CPU e as tarefas do programa; nunca períodos muito maiores do que o necessário.
- As instruções de apagamento do *watchdog* devem estar localizadas em trechos do programa freqüentemente executados. O número dessas instruções deve ser reduzido ao máximo possível.
- As rotinas de tratamento de interrupções não são, via de regra, um bom local para efetuar o apagamento da contagem do *watchdog*. Muitas aplicações podem passar a funcionar de forma anômala, mas ainda responder a interrupções.

Por fim, resta dizer que após um *reset* o *watchdog* sempre passa a operar nesse modo, utilizando como fonte de *clock* o DCO e com um período de contagem de aproximadamente 32ms.

5.22.2. Modo Temporizador

Quando operando no modo temporizador, é possível utilizar a interrupção WDTIFG para gerar interrupções em intervalos de tempo predefinidos.

Esse modo de operação é selecionado setando o *bit* WDTTMSEL no registrador WDTCTL.

Uma vez que o contador atinja a sua contagem máxima, o *flag* WDTIFG (registrador IFG1) é setado e caso a interrupção esteja habilitada (IE1:WDTIE = 1), assim como o GIE = 1, o fluxo do programa será desviado para o endereço apontado pelo vetor 10 (veja a tabela 5-1). Observe que, quando ocorre o desvio do programa para a RTI, o *flag* WDTIFG é automaticamente apagado pelo *hardware* do microcontrolador.

5.22.3. WDT+

Alguns modelos da família 4xx (42x e FE42x), além dos microcontroladores da família 2xx, incluem um circuito adicional de segurança que monitora o sinal de *clock* do *watchdog*. Nesses *chips* o *watchdog* é chamado WDT+.

No WDT+, um circuito de proteção monitora o sinal de *clock* e comuta automaticamente a sua fonte em caso de falha.

Adicionalmente, o circuito de proteção também impede a entrada em alguns modos de baixo consumo (especificamente aqueles que tentem desabilitar a fonte de *clock* do *watchdog*).

Essas funcionalidades somente estão disponíveis quando operando no modo *watchdog*.

5.22.4. Registradores do *Watchdog*

Os registradores utilizados pelo *watchdog* são os seguintes:

- **WDTCTL** - controle geral do *watchdog*;
- **IE1** - habilitação da interrupção do *watchdog*;
- **IFG1** - sinalizador da interrupção do *watchdog*.

5.22.4.1. WDTCTL

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0120	WDTCTL	Leitura	0x69							
		Escrita	WDTPW - deve sempre ser escrito 0x5A							
		Reset	0	1	1	0	1	0	0	1
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	WDT	WDT	WDTNMI	WDT	WDT	WDTSSEL	WDTISx	
		Escrita	HOLD	NMIES		TMSEL	CNTCL			
		Reset	0	0	0	0	0	0	0	0

WDTPW - senha para escrita no registrador. O valor a ser escrito deve ser sempre igual a 0x5A sob pena de o controlador gerar um *reset*.

WDTHOLD - parada do *watchdog*. Permite desativar o *watchdog* de forma e economizar energia:
0 - *watchdog* em funcionamento;
1 - *watchdog* parado (símbolo **WDTHOLD**).

WDTNMIES - seleção da borda de interrupção NMI (quando a função WDTNMI está ativada):
0 - interrupção NMI na borda de subida do pino;
1 - interrupção NMI na borda de descida do pino (símbolo **WDTNMIES**).

WDTNMI - seleção da função do pino RST/NMI:
0 - função *reset*;
1 - função NMI (símbolo **WDTNMI**).

WDTTMSEL - seleção do modo de funcionamento do *watchdog*:
0 - modo *watchdog*;
1 - modo temporizador (símbolo **WDTTMSEL**).

WDTCNTCL - apagamento da contagem do *watchdog*:
0 - contagem normal do *watchdog*;
1 - a contagem do *watchdog* é apagada (retorna a 0x0000). O *bit* WDTCNTCL também retorna a zero após a operação (símbolo **WDTCNTCL**).

WDTSSEL- seleção da fonte de *clock* do *watchdog*:
0 - *clock* secundário (SMCLK);
1 - *clock* auxiliar (ACLK) (símbolo **WDTSSEL**).

WDTISx- seleção do intervalo de contagem do *watchdog* (símbolos **WDTIS0** e **WDTIS1**):
00 - *clock*/32768;
01 - *clock*/8192;
10 - *clock*/512;
11 - *clock*/64.

5.22.4.2. IE1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0000	IE1	Leitura	UTXIE0	URXIE0	ACCVIE	NMIIE	-	-	OFIE	**WDTIE**
		Escrita								
		Reset	0	0	0	0	0	0	0	0

WDTIE - habilitação da interrupção por estouro da contagem do *watchdog* (quando ele está operando no modo *watchdog*).

5.22.4.3. IFG1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0002	IFG1	Leitura	UTXIFG0	URXIFG0	-	NMIIFG	RSTIFG	PORIFG	OFIFG	**WDTIFG**
		Escrita								
		Reset	0	0	0	0	0	0	0	0

WDTIFG - sinalizador de interrupção por estouro de contagem do *watchdog* (quando operando no modo *watchdog*).

5.22.5. Exemplos de Utilização

Vejamos alguns exemplos de configuração do *watchdog*:

Apagamento da contagem do *watchdog*:

```
MOV #WDTPW+WDTCTNCL, &WDTCTL
Ou em C:
WDTCTL = WDTPW + WDTCNTCL;
```

Alterando a fonte de *clock* do *watchdog* para o ACLK:

```
MOV #WDTPW+WDTCNTCL+SSEL, &WDTCTL
Em C:
WDTCTL = WDTPW + WDTCNTCL + SSEL;
```

Parando o *watchdog*:

```
MOV #WDTPW+WDTHOLD, &WDTCTL
Em C:
WDTCTL = WDTPW + WDTHOLD;
```

Configurando o *watchdog* para modo *timer* com intervalo igual a ACLK/512:

```
MOV #WDTPW+WDTCNTCL+WDTTMSEL+WDTSSEL+WDTIS1, &WDTCTL
Em C:
WDTCTL = WDTPW+WDTCNTCL+WDTTMSEL+WDTSSEL+WDTIS1;
```

5.23. Pino $\overline{\text{RST}}$/NMI

Os MSP430 possuem um pino dedicado às funções de *reset* externo e interrupção não-mascarável (NMI).

Quando opera na função *reset* (modo padrão após o *chip* ser ligado), um nível lógico "0" nesse pino manterá a CPU no estado de *reset*.

Quando opera na função NMI, uma transição de subida ou descida do sinal presente nesse pino seta o *flag* NMIIFG (registrador IFG1) e caso a interrupção NMI esteja habilitada (IE1:NMIIE = 1), provoca o desvio do programa para o endereço indicado no vetor 14 (veja a tabela 5-1).

A seleção da função do pino é controlada pelo *bit* WDTNMI, localizado no registrador WDTCTL, e a seleção da borda de disparo da interrupção é realizada pelo *bit* WDTNMIES, localizado no mesmo registrador.

Lembre-se de que o bit de habilitação da interrupção IE1:NMIIE é automaticamente apagado pelo hardware após a ocorrência da interrupção. Sendo assim, caso ela seja utilizada, a RTI deve providenciar a sua reativação, conforme descrito anteriormente.

5.23.1. Registradores de Controle

Os seguintes registradores são utilizados no controle do pino RST/NMI:

- WDTCTL - seleção do modo de operação do pino;
- IE1 - habilitação da interrupção NMI;
- IFG1 - sinalizador da interrupção NMI.

5.23.1.1. WDTCTL

Endereço	Nome		*BIT* 15	*BIT* 14	*BIT* 13	*BIT* 12	*BIT* 11	*BIT* 10	*BIT* 9	*BIT* 8
0x0120	WDTCTL	Leitura	0x69							
		Escrita	WDTPW - deve sempre ser escrito 0x5A							
		Reset	0	1	1	0	1	0	0	1
			BIT 7	*BIT* 6	*BIT* 5	*BIT* 4	*BIT* 3	*BIT* 2	*BIT* 1	*BIT* 0
		Leitura	WDT	WDT	WDTNMI	WDT	WDT	WDTSSEL	WDTISx	
		Escrita	HOLD	NMIES		TMSEL	CNTCL			
		Reset	0	0	0	0	0	0	0	0

WDTNMIES - seleção da borda de interrupção NMI (quando a função WDTNMI está ativada):
0 - interrupção NMI na borda de subida do pino;
1 - interrupção NMI na borda de descida do pino (símbolo **WDTNMIES**).

WDTNMI - seleção da função do pino RST/NMI:
0 - função *reset*;
1 - função NMI (símbolo **WDTNMI**).

5.23.1.2. IE1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0000	IE1	Leitura / Escrita	UTXIE0	URXIE0	ACCVIE	NMIIE	-	-	OFIE	WDTIE
		Reset	0	0	0	0	0	0	0	0

NMIIE - habilitação da interrupção NMI.

5.23.1.3. IFG1

Endereço	Nome		*BIT 7*	*BIT 6*	*BIT 5*	*BIT 4*	*BIT 3*	*BIT 2*	*BIT 1*	*BIT 0*
0x0002	IFG1	Leitura / Escrita	UTXIFG0	URXIFG0	-	NMIIFG	RSTIFG	PORIFG	OFIFG	WDTIFG
		Reset	0	0	0	0	0	0	0	0

NMIIFG - sinalizador de interrupção NMI: indica que ocorreu uma transição no sinal do pino RST/NMI.

5.24. Interfaces de Programação/Depuração

Os microcontroladores MSP430 possuem um excelente conjunto de interfaces de programação e de depuração de *software*.

Em todos os modelos estão disponíveis duas interfaces:

- JTAG, que pode ser utilizada para programação e depuração do *software* em execução no *chip*.
- BSL (*Bootstrap Loader*), que pode ser utilizada para programação do *chip* no circuito de aplicação.

No decorrer deste tópico vamos estudar as duas interfaces e conhecer alguns detalhes sobre o seu funcionamento e utilização.

5.24.1. JTAG

A interface JTAG (do acrônimo inglês ***Joint Test Action Group***) é um padrão industrial no que diz respeito a testes de circuitos eletrônicos, sendo cada vez mais utilizada pelos fabricantes de microcontroladores, microprocessores, DSPs e dispositivos lógicos programáveis, tanto para testes como para a programação e depuração de *software*.

Os dispositivos estudados neste livro utilizam basicamente quatro sinais JTAG: TDO, TDI, TMS e TCK. Além desses sinais, também é utilizado o RESET (que não é um sinal JTAG).

Nos *chips* com 20 e 28 pinos, os sinais da interface JTAG encontram-se multiplexados aos pinos de E/S. Nesses modelos, o fabricante incluiu um pino exclusivo (TEST) para acesso à interface JTAG. Um nível lógico "1" aplicado ao pino TEST faz com que os pinos compartilhados passem para o modo JTAG.

Na figura 5-52, temos o diagrama do conector padronizado da *Texas Instruments* para comunicação entre a ferramenta de depuração (chamada FET - ***Flash Emulation Tool***) e os MSP430.

O FET é um dispositivo que permite a um computador PC comunicar-se por meio da interface paralela (ou USB) com os microcontroladores MSP430.

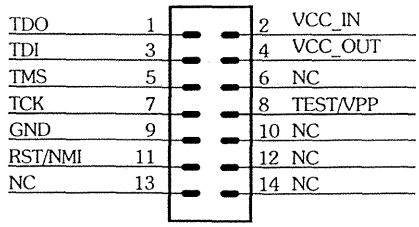


Figura 5-52

A figura 5-53 apresenta um esquema muito simples de um dispositivo capaz de realizar essa tarefa de comunicação. Esse dispositivo, batizado de MicroFET, apresenta um custo muito reduzido, ao mesmo tempo em que permite realizar as tarefas básicas de depuração e programação dos microcontroladores MSP430.

Note que, dado o seu caráter simplificado, ele não apresenta um isolamento perfeito quando utilizado com *chips* de 20 e 28 pinos, ou seja, o MicroFET pode interferir nos níveis lógicos presentes nos pinos compartilhados pela interface JTAG, mesmo quando não está em uso. Isso não chega a ser um problema em muitas aplicações, mas deve ser sempre considerado quando da sua utilização.

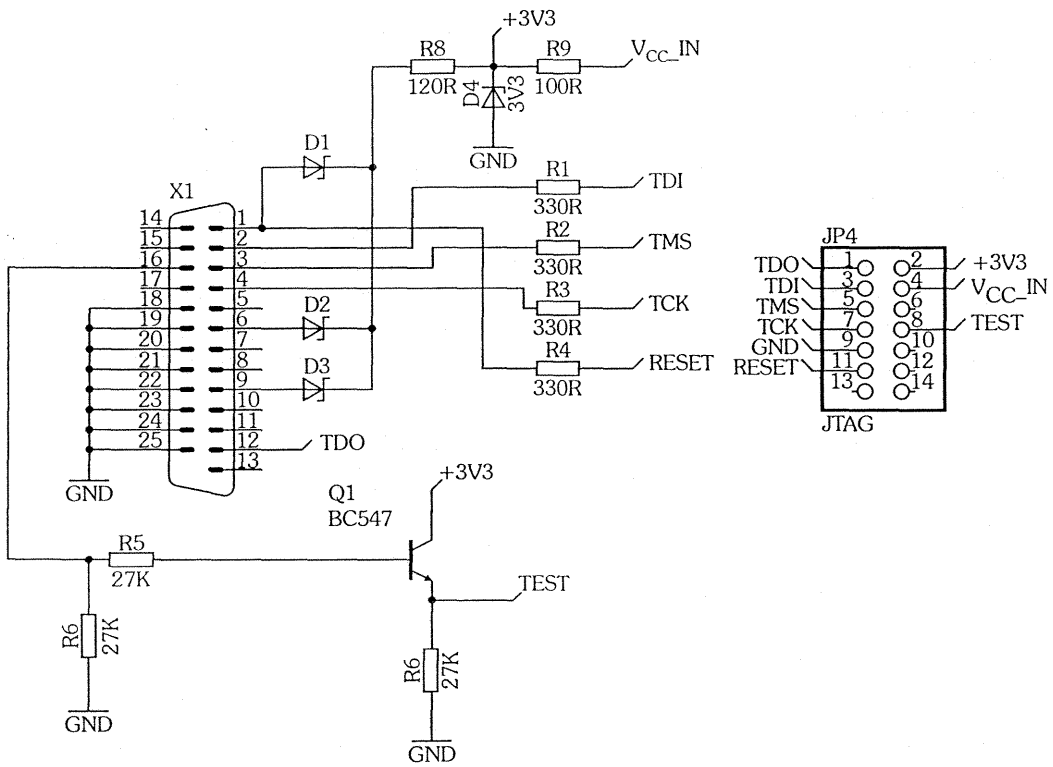


Figura 5-53

5.24.1.1. *Hardware* Interno

A interface JTAG consiste apenas no meio de comunicação entre o equipamento externo de depuração e o *hardware* interno do microcontrolador.

Para que efetivamente seja possível acompanhar e atuar sobre a execução do programa dentro do microcontrolador, é necessário que ele possua uma infra-estrutura interna preparada para isso.

Os microcontroladores MSP430 possuem um *hardware* interno de depuração capaz de:

- Iniciar e paralisar a execução do programa;
- Criar pontos de parada (*breakpoints*) no programa em execução;
- Realizar a leitura e escrita de qualquer posição da memória.

Adicionalmente, alguns dispositivos incluem controles avançados de depuração, capazes de:

- Criação de múltiplos pontos de parada;
- Paradas condicionais;
- Controle dos sinais de *clock* internos e ativação/desativação do *clock* para os periféricos na depuração.

A tabela 5-60 apresenta as características básicas do *hardware* de depuração para alguns modelos de MSP430.

O ambiente IAR inclui uma facilidade chamada de *breakpoints* virtuais: caso o programador necessite criar mais pontos de parada do que os disponibilizados pelo *hardware*, o ambiente de depuração passa a executar o programa no modo passo-a-passo automaticamente, até que o *breakpoint* definido seja atingido. Essa facilidade permite que se criem infinitos *breakpoints*, mas a um custo: a velocidade de execução cai vertiginosamente. Isso porque, utilizando apenas *breakpoints* de *hardware*, a velocidade de execução é máxima (como se não houvesse *hardware* de depuração conectado) e o programa pára somente ao atingir um *breakpoint*. No caso dos *breakpoints* virtuais, o *hardware* executa uma instrução e pára, recebe um comando, executa a próxima instrução, pára novamente e assim por diante, até que seja atingido um dos *breakpoints* virtuais programados.

Modelo	*Breakpoints*	Depuração Avançada
F11xx	2	N
F12x	2	N
F12xx	2	N
F13x	3	N
F14x	3	N
F15x	8	S
F16x	8	S
F41x	2	N
F42x	2	N
F43x	3	S
F44x	8	S

Tabela 5-60

5.24.1.2. Proteção de Acesso

A interface JTAG do MSP430 inclui um fusível de proteção que pode ser utilizado para impedir o acesso ao programa gravado na memória do microcontrolador.

O fusível de proteção é testado pelo *hardware* interno do microcontrolador sempre que se realiza um acesso pela interface JTAG. Estando intacto, o acesso é permitido. Caso o fusível tenha sido programado ou popularmente "queimado", o acesso não é permitido, não sendo mais possível executar nenhum comando de apagamento, programação ou depuração pela interface.

O procedimento para programação ou "queima" desse fusível somente pode ser realizado pela própria interface JTAG e consiste nos seguintes passos:

1. O computador ou programador envia o comando JTAG de preparação para "queima" do fusível e em seguida o comando de "queima" do fusível.
2. Aplica-se uma tensão entre 6 e 7 Volts no pino TDI/TCLK.

   O processo de "queima" do fusível pode necessitar de até 1ms para ser concluído. Durante esse período, a corrente consumida no pino TDI/TCLK pode chegar a até 100mA.

   A maioria dos *hardwares* de depuração (FET) disponíveis não permite que seja efetuada a programação do fusível de proteção da interface JTAG, no entando a *Texas Instruments* fornece ferramentas de baixo custo para programação do microcontrolador e do fusível de proteção, entre eles o MSP-PRGS430 *Serial Programmer* e o MSP430 *In-System Gang Programmer*. Maiores detalhes podem ser obtidos no *site* do fabricante.

Lembre-se de que, programado o fusível de proteção, nenhum acesso pela interface JTAG é permitido. Essa operação não pode ser desfeita e a única forma de reprogramar o chip é pelo BSL!

5.24.2. *Bootstrap Loader* (BSL)

Outra modalidade de programação do microcontrolador é o *Bootstrap Loader* (BSL), que consiste em um *software* gravado em uma porção de memória ROM disponível em todos os modelos de MSP430 (endereço 0x0C00 a 0x0FEF).

Esse *software* é automaticamente executado quando se aplicam, no mínimo, dois pulsos no pino TCK (*chips* com mais de 28 pinos) ou TEST (*chips* com 28 pinos ou menos), enquanto o pino de *reset* (RST/NMI) é mantido em nível lógico baixo. A transição do nível lógico "0" para "1" no pino RST/NMI faz com que o *chip* saia do *reset* e passe a executar o BSL.

Executando o BSL, a comunicação é feita serialmente com o microcomputador, a uma velocidade de 9600 bps, no formato 8E1 (8 *bits* de dados, paridade par e 1 *bit* de parada).

O BSL pode receber um dos seguintes comandos:

1. Recepção de senha;
2. Apagamento geral;
3. Envio da versão do BSL (somente a partir da versão 1.60 do BSL ou utilizando a versão em RAM, conforme veremos mais adiante);

4. Mudança de velocidade de comunicação. A velocidade de comunicação pode ser selecionada entre 9.600, 19.200 ou 38.400 bps (somente a partir da versão 1.60 do BSL ou utilizando a versão em RAM, conforme veremos mais adiante);
5. Recepção de um bloco de dados (do PC para o MSP430) para programação da *FLASH* ou escrita em uma posição de memória qualquer;
6. Transmissão de um bloco de dados (do MSP430 para o PC), permitindo que o conteúdo da memória do *chip* seja lido e armazenado no computador *host*;
7. Apagamento de um segmento de memória;
8. Verificação de apagamento (somente a partir da versão 1.60 do BSL ou utilizando a versão em RAM, conforme veremos mais adiante);
9. Execução do programa a partir de um endereço especificado.

Uma característica muito importante do BSL é que a maioria dos comandos relacionados é protegida por uma senha de 256 *bits* que deve ser enviada previamente utilizando o comando [1]. Os comandos [1] a [4] não necessitam de senha para a sua execução.

A senha está localizada na área de vetores de interrupção dos MSP430 (endereço 0xFFE0 a 0xFFFF).

Nos microcontroladores da família 2xx, como medida de segurança adicional, caso a senha fornecida seja incorreta, a memória do *chip* é automaticamente apagada, ressetando a senha (todos os *bytes* iguais a 0xFF).

Outra característica interessante do BSL está na possibilidade de atualizar o seu código. Claro que se o código estiver armazenado em ROM, não é possível alterar a versão do BSL gravada no *chip*, mas é possível carregar uma nova versão em RAM e executá-la daquele ponto.

O fabricante recomenda que, sempre que for possível, se utilizem as versões mais recentes do BSL, disponíveis para *download* no *site* www.ti.com.

No lado do computador *host* (PC), executa-se um programa que efetua a conexão com o BSL por uma das portas seriais disponíveis. Esse programa pode ser escrito pelo usuário, ou ainda ser baixado do *site* da *Texas Instruments*.

Maiores detalhes sobre os comandos, funcionamento detalhado e utilização do BSL podem ser obtidos nos documentos SLAA089B e SLAA096B (referências bibliográficas 22 e 23).

5.24.2.1. Conexões do BSL

O BSL utiliza os seguintes pinos para o seu funcionamento:

Sinal	Pino
RST/NMI	RST/NMI
TCK	TCK (somente *chips* com mais de 28 pinos)
TEST	TEST (*chips* com 28 pinos ou menos)
BSL_TX	P1.1 (famílias 1xx e 2xx) P1.0 (família 4xx)
BSL_RX	P2.2 (famílias 1xx e 2xx) P1.1 (família 4xx)

Tabela 5-61

Lembre-se de que os pinos utilizados pelo BSL não podem ser conectados diretamente a uma interface EIA232. É necessário utilizar um circuito conversor de nível para esta finalidade. A nota de aplicação SLAA096B disponível no *site* do fabricante apresenta um circuito que pode ser utilizado para essa tarefa.

5.24.2.2. Senha do BSL

É importante programarmos os vetores de interrupção não utilizados, de forma a aumentar o grau de segurança da senha do BSL. Isto porque quando desprogramados, os vetores contêm o valor 0xFFFF, fazendo com que este seja o primeiro valor a ser testado por um eventual "*cracker*" (termo utilizado para aqueles que tentam "quebrar" ou desvendar senhas de sistemas protegidos).

Num *chip* onde não seja utilizada nenhuma interrupção , o único vetor utilizado é o de *reset*. Neste caso, podemos programar os vetores não utilizados criando uma matriz com os valores desejados.

```
__root const unsigned char senha[] @ 0xFFE0 =
                {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF, 0x01,
                 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF, 0x01, 0x23,
                 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF, 0x01, 0x23, 0x45,
                 0x67, 0x89, 0xAB};
```

Ou:

```
__root const unsigned int senha[] @ 0xFFE0 =
        { 0x2301, 0x6745, 0xAB89, 0xEFCD, 0x2301,
          0x6745, 0xAB89, 0xEFCD, 0x2301, 0x6745,
          0xAB89, 0xEFCD, 0x2301, 0x6745};
```

Caso alguma interrupção seja utilizada, o programador deve arranjar a matriz acima de forma a não sobreescrever o vetor da interrupção efetivamente utilizada. Por exemplo, se estiver utilizando apenas a interrupção da porta 2, em um *chip* MSP430F149, poderíamos programar os vetores restantes com a seguinte declaração:

```
__root const unsigned int senha[] @ 0xFFE4 =
        { 0x2301, 0x6745, 0xAB89, 0xEFCD, 0x2301,
          0x6745, 0xAB89, 0xEFCD, 0x2301, 0x6745,
          0xAB89, 0xEFCD, 0x2301};
```

A área de vetores de interrupção, na memória do microcontrolador, será programada com os seguintes valores:

Endereço	Valor
0xFFE0	0xFFFF
0xFFE2	Vetor da interrupção da porta 2
0xFFE4	0x2301
0xFFE6	0x6745
0xFFE8	0xAB89
0xFFEA	0xEFCD
0xFFEC	0x2301
0xFFEE	0x6745
0xFFF0	0xAB89
0xFFF2	0xEFCD
0xFFF4	0x2301
0xFFF6	0x6745

Endereço	Valor
0xFFF8	0xAB89
0xFFFA	0xEFCD
0xFFFC	0x2301
0xFFFE	Vetor de reset

Tabela 5-62

Devido a forma como o compilador aloca os segmentos de memória dos vetores de interrupção e das constantes, utilizando a técnica acima, somente podemos escrever após o último vetor ocupado, excluíndo-se o de *reset*. Por isso, o primeiro endereço (0xFFE0) não foi programado, enquanto que o segundo (0xFFE2) irá possuir o valor referente ao endereço da RTI da porta 2.

5.24.3. Depuração pela Interface JTAG

O procedimento para depuração *in-circuit* utilizando a interface JTAG é muito simples e praticamente idêntico ao descrito para a simulação do programa no capítulo 3.

Para utilizar o depurador em vez do simulador, é necessário abrir a janela de opções do projeto (*Project* > *Options*, ou o atalho ALT+F7) e em seguida selecionar a opção *Debugger* (figura 5-54). Na aba *Setup*, deve ser selecionado *o driver FET Debugger*.

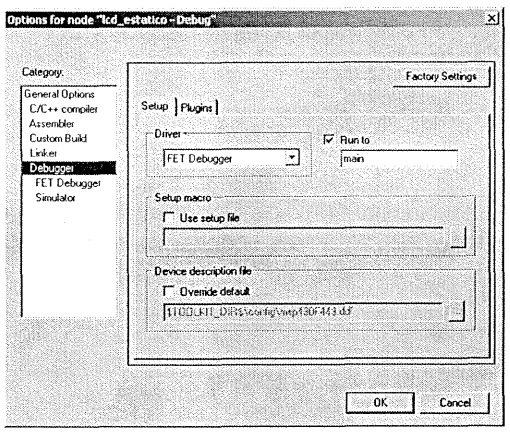


Figura 5-54

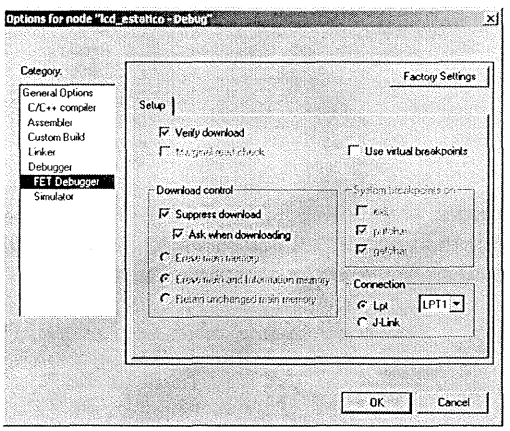


Figura 5-55

Clicando na categoria *FET Debugger*, podemos configurar algumas opções do depurador, entre elas o tipo de conexão (paralela ou USB), a verificação do programa após a gravação e o controle de programação, que permite efetuar a comunicação com o *chip* sem programá-lo (opção *Supress download*).

Ao clicar no botão DEBUG, o computador tenta estabelecer contato com o *chip* e caso seja bem-sucedido, o depurador será carregado surgindo a mesma tela da figura 3-16.

6

Programação em C

6.1. Revisão da Linguagem C

Antes de estudarmos os detalhes do compilador IAR, faremos uma rápida revisão dos pontos mais importantes da linguagem C. Caso o leitor sinta-se suficientemente familiarizado com a linguagem (padrão ANSI), pode pular para o tópico 6.2.

6.1.1. Comandos e Palavras Reservadas

A linguagem C possui um conjunto de 32 palavras reservadas que constituem os comandos e declarações possíveis dentro da linguagem e não podem ser redefinidas dentro do programa.

auto	*break*	*case*	*char*	*const*
continue	*default*	*do*	*double*	*else*
enum	*extern*	*float*	*for*	*goto*
if	*int*	*long*	*register*	*return*
short	*signed*	*sizeof*	*static*	*struct*
switch	*typedef*	*union*	*unsigned*	*void*
volatile	*while*			

Tabela 6-1

6.1.2. Estrutura Básica de um Programa

A estrutura básica de um programa em linguagem C pode ser exemplificada como na listagem em seguida.

```
// Inclusão de arquivos externos
// Declaração de variáveis globais
// Declaração de protótipos de funções
// Funções
// Função principal (main)
```

De todos os passos anteriores, qualquer programa em C deve possuir, ao menos, a função principal (ou *main* em inglês) que contém o corpo principal do programa e é a primeira função a ser executada quanto o programa é iniciado.

Em seguida, temos um exemplo de escrita de uma função principal:

```
void main(void)
{
     char a,b;
     a = 10;
     b = a + 5;
     // comentário 1
     /*
        comentário 2
        fim do comentário 2
     */
}
```

Repare que os comandos pertencentes à função principal (*main*) estão escritos dentro de um bloco de comandos, delimitado pelo símbolo de abertura "{" e fechamento "}" de chaves.

Um bloco de comandos é utilizado para declarar os comandos a serem executados pela função, ou para declarar uma seqüência de comandos a serem executados por outros comandos condicionais da linguagem C.

Um outro aspecto importante da linguagem C é o terminador do comando, que consiste em um sinal de ponto-e-vírgula ";".

O primeiro comando dentro da função principal anterior é **char** a,b que consiste em uma declaração de duas variáveis, chamadas "a" e "b".

Em seguida, temos uma atribuição simples do valor 10 decimal à variável "a".

Finalmente, o último comando consiste em uma expressão que atribui o resultado da soma do valor de "a" (que é 10) com 5 à variável "b", ou seja, "b" recebe o valor 15 decimal.

Observe que a base numérica padrão em C é a decimal. Para escrever valores em hexadecimal, utiliza-se o prefixo "0x". Assim, o valor 15 decimal poderia ser escrito 0x0F.

Após o último comando, temos dois exemplos de como podem ser feitos comentários dentro de um programa em C: o primeiro utilizando os caracteres "//" é chamado de comentário de linha simples. O compilador considera tudo o que segue os caracteres "//" como um comentário, até o final daquela linha. Esse tipo de comentário é utilizado para documentar linhas de código individuais.

O segundo tipo de comentário utiliza os caracteres "/*" para delimitar o início do comentário e "*/" para delimitar o fim do comentário. Este é o chamado comentário de múltiplas linhas e pode ser utilizado para escrever grandes trechos de comentários. É muito utilizado para documentação de funções e do programa de forma geral.

6.1.3. Tipos de Dados

A linguagem C dispõe de basicamente três tipos de dados: inteiros, reais e nulo. Os dados de tipos inteiros podem ainda ser sinalizados (positivos ou negativos) ou não-sinalizados (somente positivos). Os dados reais podem representar tanto valores inteiros como fracionários. O tipo nulo (***void***) é utilizado para declaração de funções que não retornam e/ou não recebem valores (parâmetros) e também para declaração de ponteiros genéricos.

A linguagem C prevê ainda o uso de modificadores de tipo:

short - para reduzir a faixa de representação do tipo;

long - para ampliar a faixa de representação do tipo;

signed - para especificar a representação de valores sinalizados (utilizando o formato de complemento de dois);

unsigned - para especificar a representação de valores não-sinalizados.

O padrão ANSI considera que um tipo base é sempre ***signed*** (com sinal), a não ser que seja expressamente declarado como ***unsigned*** (sem sinal). A exceção é o tipo ***char***, deixado em aberto pela norma ANSI. No compilador IAR, o tipo ***char*** é, por padrão, ***unsigned***, a não ser que seja declarado como ***signed***.

Considerando os tipos básicos e os subtipos formados com o emprego dos modificadores anteriores, temos os seguintes tipos de dados em linguagem C:

Tipo	Faixa de valores	Tamanho em *Bytes*
unsigned char, char	0 a 255	1
signed char	–128 a 127	1
unsigned int, unsigned short int	0 a 65.535	2
int, signed int, short int, signed short int	–32.768 a 32.767	2
unsigned long int	0 a 4.294.967.295	4
long int, signed long int	–2.147.483.648 a 2.147.483.647	4
float	$1{,}18 \times 10^{-38}$ a $3{,}4 \times 10^{+38}$	4
double	$2{,}23 \times 10^{-308}$ a $1{,}8 \times 10^{+308}$	8
long double	$2{,}23 \times 10^{-308}$ a $1{,}8 \times 10^{+308}$	8

Tabela 6-2

Os seis primeiros tipos listados na tabela 6-2 são inteiros, enquanto os três últimos são tipos reais (também chamados de ponto flutuante em razão do formato de representação utilizado).

6.1.3.1. Declaração de Variáveis

As variáveis em C devem ser declaradas sempre no início do programa, de uma função ou de um bloco de programa.

O identificador (nome) da variável pode ter qualquer tamanho (normalmente são reconhecidos os 31 primeiros caracteres) e deve obedecer às seguintes regras:

- O identificador deve conter somente letras, números e o caractere sublinhado "_". Isso significa que não é permitido utilizar símbolos ou acentos nos identificadores.
- O identificador deve iniciar por uma letra ou sublinhado.
- Letras maiúsculas e minúsculas são diferentes em linguagem C (isso significa que o identificador `TESTE` é diferente do identificador `teste`.
- Um identificador não pode ser igual a uma palavra reservada ou um nome de função definida no programa.

Isto posto, a declaração de variáveis em C é feita da seguinte forma:

```
int teste;  // declara uma variável inteira chamada teste
unsigned int minha_variavel; // declara uma variável inteira em sinal
char primeiro, segundo;  // declara duas variáveis do tipo char
float _resultado; // declara uma variável real chamada _resultado
```

Um outro aspecto importante sobre a declaração de variáveis diz respeito ao local em que ela foi declarada. Existem dois locais básicos onde é possível declarar variáveis:

No início do programa, antes de qualquer função: são as chamadas variáveis globais e podem ser acessadas de qualquer ponto do programa.

No início de uma função ou bloco de programa: são as chamadas variáveis locais e somente podem ser acessadas dentro da função ou bloco de programa no qual foram declaradas. Além disso, são destruídas após a saída do bloco ou da função.

Um tipo especial de variável local são os parâmetros formais de uma função, que comportam-se como variáveis locais, mas são declarados junto com a função. Esses parâmetros formais são utilizados pelo compilador para a passagem de parâmetros para a função.

6.1.3.2. Outros Modificadores

Além dos modificadores de tipo já comentados, a linguagem C dispõe de outros com características mais específicas:

auto - especifica uma variável automática ou local. Esse especificador já não é mais utilizado, pois todas as variáveis declaradas dentro de uma função ou bloco de programa são consideradas locais, independente do seu uso.

const - utilizado para declarar uma constante (variável cujo valor não pode ser alterado pelo programa). Tais variáveis podem ser armazenadas na memória *FLASH*.

extern - utilizado para fazer referências a variáveis globais declaradas em outros módulos do programa.

register - solicita ao compilador que tente armazenar a variável em um registrador da CPU, em vez de uma posição de memória RAM. Isso permite um incremento na velocidade de acesso à variável, porém esse especificador somente pode ser utilizado para variáveis locais ou parâmetros formais de uma função.

static - utilizado para declarar variáveis permanentes, ou seja, que não são destruídas após a saída da função ou bloco de programa em que foram declaradas: o seu conteúdo permanece inalterado e na próxima chamada da função, ou execução do bloco de comandos, o seu valor continuará o mesmo da última chamada. Esse especificador é normalmente utilizado com variáveis locais. No caso da declaração de uma variável global, ela somente será acessível de dentro do módulo (listagem) do programa em que foi declarada. Uma variável declarada como ***static*** é sempre inicializada com zero pelo compilador.

volatile - especifica uma variável que pode ser alterada por eventos externos sem conhecimento do compilador. Esse tipo de especificador deve ser utilizado para variáveis que possam ser alteradas pelo *hardware* do sistema.

6.1.4. Operadores e Expressões em C

Operadores são elementos utilizados para realizar uma operação (aritmética, lógica, relacional ou outra qualquer) com um ou dois elementos. A linguagem C possui um amplo e completo conjunto de operadores que se encontram listados na tabela 6-3.

Operador	Ação	Função
=	Atribuição	O operando da esquerda recebe o valor do operando ou resultado da expressão à direita
+	Adição	Adiciona dois operandos
–	Subtração ou menos unário	Subtrai dois operandos, ou torna o operando negativo
*	Multiplicação	Multiplica dois operandos
/	Divisão	Divide dois operandos
%	Resto de divisão inteira	Retorna o resto da divisão de dois operandos inteiros
++	Incremento	Aumenta um no conteúdo do operando
--	Decremento	Diminui um no conteúdo do operando
&	Operação lógica E *bit* a *bit*	Realiza a operação lógica E entre os *bits* dos operandos
|	Operação lógica OU *bit* a *bit*	Realiza a operação lógica OU entre os *bits* dos operandos
^	Operação lógica OU Exclusivo *bit* a *bit*	Realiza a operação lógica Exclusivamente OU entre os *bits* dos operandos
~	Complemento de um (inverte todos os *bits*)	Realiza a operação lógica NÃO nos *bits* do operando
>>	Deslocamento à direita	Desloca os *bits* do operando "n" posições à direita
<<	Deslocamento à esquerda	Desloca os *bits* do operando "n" posições à esquerda
!	Operação lógica NÃO booleana	Retorna verdadeiro caso o operando seja falso
&&	Operação lógica E booleana	Retorna verdadeiro caso ambos os operandos sejam verdadeiros
||	Operação lógica OU booleana	Retorna verdadeiro caso um dos operandos, ou os dois sejam verdadeiros
>	Maior que (relacional)	Retorna verdadeiro se o operando da esquerda é maior que o da direita
>=	Maior ou igual a (relacional)	Retorna verdadeiro se o operando da esquerda é maior ou igual ao operando da direita
<=	Menor ou igual a (relacional)	Retorna verdadeiro se o operando da esquerda é menor ou igual ao operando da direita
<	Menor que (relacional)	Retorna verdadeiro se o operando da esquerda é menor que o operando da direita
==	Igual a (relacional)	Retorna verdadeiro se os operandos da esquerda e da direita são iguais
!=	Diferente de (relacional)	Retorna verdadeiro se o operando da esquerda é diferente do operando da direita
&	Operador unário endereço de (ponteiro)	Retorna o endereço do operando (que contém uma variável ou nome de função)
*	Operador unário valor de (ponteiro)	Retorna o valor do operando (que contém um endereço)
.	Operador de referência a elemento de estrutura ou união	Retorna um elemento da estrutura ou união
->	Operador de ponteiro para elemento de estrutura	Retorna um elementos da estrutura ou união
?	Operador de expressão condicional	Avalia a primeira expressão e caso a mesma seja verdadeira, retorna o resultado da segunda expressão. Caso seja falsa, retorna o resultado da terceira expressão
,	Operador de encadeamento de expressões	Realiza uma seqüência de operações dentro de uma atribuição
( )	Operador de priorização de operação	Prioriza a operação delimitada pelos mesmos
[ ]	Operador de índice de matriz	Indica um elemento pertencente a uma matriz de dados
sizeof	Operador de tempo de compilação para verificação de espaço ocupado	Retorna o tamanho em *bytes*, ocupado pelo operando especificado na memória do sistema

Tabela 6-3

É possível ainda associar alguns operadores, originando novos operadores:

Forma reduzida	Forma original
x += y	x = x + y
x –= y	x = x – y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x &= y	x = x & y
x |= y	x = x | y
x ^= y	x = x ^ y
x <<= y	x = x << y
x >>= y	x = x >> y

Tabela 6-4

Para construir expressões, utilizamos um ou mais operadores. A ordem de avaliação da expressão é da esquerda para a direita. No entanto, é permitido que o compilador rearranje a expressão de forma a otimizar o código gerado (desde que não se altere o resultado da expressão, é claro). Sendo assim, o programador não deve contar que a expressão será processada exatamente na ordem em que foi escrita.

A ordem de prioridade dos operadores no processamento de uma expressão é determinada pela tabela 6-5.

Ordem	Operador
Maior	( ) [ ] –>
	! ~ ++ — . (tipo) * & sizeof
	* / %
	+ –
	<< >>
	<<= >>=
	== !=
	&
	^
	|
	&&
	||
	?
	= += –= *= /=
Menor	,

Tabela 6-5

Outra característica da linguagem C e que é utilizada no processamento das expressões, a chamada promoção de tipo, e consiste num conjunto de regras aplicadas pelo compilador com o intuito de que todos os elementos da expressão sejam convertidos no mesmo tipo. As regras estão expostas em seguida:

1. Numa expressão contendo dois operandos de tipos diferentes, o tipo de menor tamanho é convertido no tipo de maior tamanho. A ordem dos tipos, para efeitos de promoção de tipo é (do menor para o maior): ***char***, ***short***, ***int***, ***long***, ***float***, ***doublé.***
2. Numa expressão contendo um tipo ***unsigned*** e outro ***signed***, o ***signed*** é convertido em ***unsigned.***

É possível também forçar uma conversão de tipo (*typecast*), utilizando o especificador do tipo desejado entre parênteses antes da variável ou expressão que se deseja converter.

6.1.5. Tipos de Dados Complexos

Além dos tipos básicos, a linguagem C permite também a construção de outros tipos de dados mais complexos.

6.1.5.1. Enumerações

Enumerações é um conjunto de constantes inteiras utilizado para especificar quais os valores que uma variável pode assumir.

A declaração de uma enumeração é feita geralmente da seguinte forma:

```
enum nome_da_enumeração
{
        lista_de_identificadores
} variável_enumerada
```

Cada elemento da lista de identificadores é separado do anterior por uma vírgula ",". Normalmente, o primeiro elemento da lista é o de valor 0, o segundo o de valor 1 e assim por diante, porém é possível especificar a qualquer tempo o número de ordem do elemento, bastando utilizar o operador "=" seguido do valor de enumeração desejado.

Repare que um elemento da lista recebe sempre o valor de enumeração do elemento anterior mais um. Sendo assim, se alterarmos o valor de enumeração de um dos elementos, os posteriores a ele também terão seus valores alterados.

6.1.5.2. Ponteiros

Os ponteiros, ou apontadores, estão entre os grandes destaques que tornam a linguagem C tão poderosa e consistem simplesmente em variáveis que armazenam endereços de memória onde, normalmente, encontram-se variáveis.

Os operadores & e * são utilizados para as operações básicas dos ponteiros: & retorna o endereço de um variável, enquanto * retorna o conteúdo de um endereço especificado.

É possível declarar variáveis para guardar ponteiros. Neste caso, declara-se a variável normalmente, mas utilizando o operador * antes do seu nome:

```
char A;     // variável tipo char chamada A
char *A; // variável chamada A que aponta para um char
```

É possível também utilizar os operadores de adição, subtração, incremento e decremento com variáveis ponteiro, no entanto o programador deve ter em mente que como um ponteiro deve sempre apontar para uma variável do tipo para o qual foi declarado, as operações serão sempre

realizadas levando em conta o tamanho em *bytes* ocupado pelo tipo apontado pelo ponteiro, por exemplo: se um ponteiro para um ***char*** que aponta para o endereço x é incrementado, o seu conteúdo passará a ser x+1. Se o ponteiro fosse para um ***int***, o incremento resultaria no endereço x+2. Se o ponteiro fosse para um ***double***, o incremento resultaria no endereço x+8. Isso tudo porque um ***char*** ocupa um *byte* na memória, enquanto um ***int*** ocupa dois e um ***double*** ocupa normalmente oito *bytes*.

Uma outra característica dos ponteiros é que é possível utilizar um ponteiro para uma função. Neste caso, o endereço armazenado no ponteiro é o do início do código da função na memória. Para atribuir o endereço de uma função a um ponteiro, basta utilizar o nome da função (sem os parênteses ou argumentos).

6.1.5.3. Matrizes

Matrizes são arranjos seqüenciais e contínuos de elementos de um mesmo tipo de dado.

A linguagem C permite a construção de matrizes unidimensionais (listas) ou multidimensionais (tabelas) utilizando quaisquer dos tipos de dados permitidos em C (inclusive ponteiros e outros tipos de dados complexos).

A declaração de uma matriz pode ser feita fornecendo o número máximo de elementos desejados para cada dimensão:

```
// declara uma matriz unidimensional de 25 elementos char chamada "matriz1"
char matriz1 [25];
// declara uma matriz unidimensional de 10 elementos do tipo unsigned int
// chamada "matriz2"
unsigned int matriz2 [10];
// declara uma matriz bidimensional de 5 elementos em cada dimensão, do tipo
// float chamada "matriz3"
float matriz3 [5] [5];
```

Também é possível declarar e inicializar os elementos de uma matriz:

```
// declara uma matriz unidimensional de char chamada "notas" e inicializada
// com os valores apresentados
char notas [5] = {4, 8, 6, 8, 9};
```

Repare que para fazer referência a um elemento da matriz, utilizamos o identificador da matriz seguido do número do elemento dentro dos colchetes. Assim, para atribuir o valor da primeira nota a uma variável ***char*** chamada primeira_nota faríamos:

```
primeira_nota = notas [0];
```

Observe que o primeiro elemento da matriz "notas" anterior é o de índice zero e o último o de índice 4. **Não há um elemento notas [5] !!!!**

É possível também acessar um elemento de uma matriz por ponteiros: em C, o identificador da matriz sem qualquer referência ao elemento dela aponta para o endereço do primeiro elemento da matriz. Isso significa que as declarações em seguida produzem o mesmo resultado:

```
notas [1] = 10; // o segundo elemento de notas é igual a 10
*(notas + 1) = 10;      // o segundo elemento de notas é igual a 10
```

6.1.5.4. Estruturas

Outro tipo de dado complexo admitido em C são as estruturas. Uma estrutura é um agrupamento de variáveis individuais de qualquer tipo, referenciadas por um nome comum. As variáveis pertencentes a uma estrutura são denominadas campos da estrutura.

A declaração de uma estrutura tem o seguinte formato:

```
struct nome_da_estrutura
{
      tipo campo1;
      tipo campo2;
      ...
} variáveis_estrutura;
```

Observe que a declaração de um nome para a estrutura é opcional, assim como a declaração imediata de variáveis utilizando a estrutura.

Se o programador desejar, pode declarar a estrutura sem um nome, desde que declare imediatamente as variáveis utilizando a estrutura.

Por outro lado, é possível declarar uma estrutura com nome, mas sem declaração imediata de variáveis. Neste caso, é possível declarar posteriormente variáveis utilizando o nome da estrutura. Veja o exemplo em seguida, no qual declaramos uma estrutura de horário, contendo três campos: hora, minuto e segundo:

```
struct ehorario
{
     char hora;
     char minuto;
     char segundo;
} horario;
...
...
// declara uma variável de estrutura chamada "alarme" do tipo ehorario
struct ehorario alarme;
```

A referência a um elemento de uma estrutura é feita utilizando o identificador da variável de estrutura seguido de um ponto "." e a identificação do campo. Para atribuirmos o valor 10:30:00 à estrutura "horario", poderíamos fazer:

```
horario.hora = 10;
horario.minuto = 30;
horario.segundo = 0;
```

Também é possível acessar um elemento de uma estrutura por um ponteiro para a estrutura. Neste caso, utilizamos o operador seta "->" para especificar o campo desejado:

```
ponteiro_para_estrutura->campo = valor;
```

Uma outra facilidade adicional permitida em linguagem C são os campos de *bit*, que consistem em campos especiais de estrutura, cujo número de *bits* ocupado pode ser especificado pelo programador. Campos de *bit* são muito úteis para acesso a *bits* individuais de uma variável, ou registrador do microcontrolador.

A declaração de um campo de *bit* é feita da mesma forma que a declaração de um campo qualquer de estrutura, com a diferença de que após o nome do campo, o programador deve inserir um sinal ":" e em seguida o número de *bits* utilizado pelo campo.

A referência a um campo de *bit* é feita da mesma forma que em uma estrutura normal.

Em seguida temos um exemplo da declaração de uma estrutura para o registrador P1DIR. Cada um dos *bits* possui um tamanho de campo igual a 1.

```
struct
{
   unsigned char P1DIR_0        : 1;
   unsigned char P1DIR_1        : 1;
   unsigned char P1DIR_2        : 1;
   unsigned char P1DIR_3        : 1;
   unsigned char P1DIR_4        : 1;
   unsigned char P1DIR_5        : 1;
   unsigned char P1DIR_6        : 1;
   unsigned char P1DIR_7        : 1;
} P1DIR_bit;
```

O acesso a um determinado *bit* da estrutura pode ser feito por uma simples declaração:

```
P1DIR_bit.P1DIR_0 = 1;  // configura o pino P1.0 como saída
```

6.1.5.5. Uniões

Uniões são um tipo especial de dado no qual as variáveis declaradas residem num mesmo endereço da memória.

O compilador aloca memória para a variável cujo tipo ocupe o maior espaço na memória e em seguida, as demais variáveis pertencentes à união são alocadas no mesmo espaço.

O compilador alinha as variáveis pertencentes a uma união pelos seus dígitos mais significativos (modelo *little endian*).

A declaração de uma união é bastante similar a de uma estrutura. Vejamos um exemplo:

```
union
{
      unsigned long int var32;
      unsigned int var16;
      unsigned char var8;
} tipos;
```

Se efetuarmos a atribuição: tipos.var32 = 0x12345678, teremos os seguintes valores nos demais campos: tipos.var16 = 0x1234 e tipos.var8 = 0x12. Isso porque todos os campos compartilham o mesmo endereço inicial de armazenamento. A tabela 6-6 ilustra este fato.

Campo	Endereço na memória			
	Endereço	Endereço+1	Endereço+2	Endereço+3
var32	0x12	0x34	0x56	0x78
var16	0x12	0x34		
var8	0x12			

Tabela 6-6

Um detalhe importante sobre uniões é que elas não podem ser passadas como um parâmetro de uma função.

6.1.5.6. Definições de Tipos

Outra característica interessante da linguagem C, mas que não consiste realmente em um novo tipo de dado, é o comando ***typedef***.

Ele permite criar novos identificadores para tipos de dados já existentes. Isso facilita a legibilidade de alguns programas e também permite tornar alguns tipos de programas escritos para outras plataformas mais facilmente portáveis.

A utilização de ***typedef*** é bastante simples. Supondo que seja desejado dar o nome de "fracionário" para o tipo de dado ***float***, bastaria o seguinte comando:

```
typedef float fracionario;
```

Em seguida, poderíamos declarar uma nova variável como:

```
fracionario teste;      // declara uma variável chamada teste do
                        // tipo fracionario (float)
```

6.1.6. Comandos da Linguagem C

A linguagem C possui um conjunto de comandos de controle de programa organizados nas seguintes categorias:

- Condicionais: são comandos que permitem executar código associado a uma condição ser verdadeira ou falsa.
- Iteração, laço ou repetição: permitem a execução de código repetitivo.
- Salto ou desvio: utilizados para provocar o desvio do fluxo do programa.

Existem dois tipos de comandos condicionais em C: ***if*** e ***switch***.

6.1.6.1. *If*

O comando ***if*** permite executar um comando ou bloco de comandos, caso uma condição seja verdadeira e opcionalmente, um outro comando ou bloco de comandos, no caso de a condição ser falsa.

Lembre-se de que para a linguagem C verdadeiro consiste em qualquer valor diferente de zero e falso consiste no valor zero.

A sintaxe do comando ***if*** é:

```
if (condição) comando_caso_verdadeiro; else comando_caso_falso;
```

Repare que a cláusula ***else*** e o seu comando associado são opcionais.

Alternativamente ao ***if***, é possível utilizar o operador ternário "?" nos casos em que o ***if*** controla uma atribuição condicional.

6.1.6.2. *Switch*

O comando ***switch*** permite selecionar uma opção entre várias de acordo com o conteúdo de uma variável ou expressão. Sua sintaxe é a seguinte:

```
switch (expressão)
{
     case constante1 :
                        comando;
                        comando;
                        break;
     case constante2 :
                        comando;
                        comando;
                        break;
     case constanteN :
                        comando;
                        ...
                        break;
     default :
                        comando;
                        ...
}
```

O seu funcionamento é o seguinte: o resultado da expressão é comparado com cada uma das constantes associadas às cláusulas ***case*** e quando a comparação resulta verdadeira, os comandos daquela cláusula são executados até que seja encontrado um ***break***.

Caso nenhuma das comparações resulte verdadeira, os comandos associados à cláusula ***default*** são executados. No entanto, a inclusão dessa cláusula é opcional.

Lembre-se de que duas constantes ***case*** no mesmo ***switch*** não podem ter o mesmo valor.

Vejamos agora os comandos de iteração ***for***, ***while*** e ***do***:

6.1.6.3. *For*

O comando ***for*** consiste num comando de iteração ou repetição condicional, ou seja, ele é utilizado para executar um comando ou bloco de comandos, enquanto uma condição for verdadeira.

Sua sintaxe é:

for (inicialização; condição; incremento) comando;

O seu funcionamento é o seguinte:

1. Inicialmente é executada a operação expressa na seção "inicialização" do comando.
2. A seção "condição" é testada e se for verdadeira, o comando ou bloco de comandos associado ao ***for*** é executado.
3. É executada a seção "incremento" do comando ***for.***
4. A execução retorna para o item 2 e permanecerá nesta repetição até que a "condição" avaliada seja falsa, o que encerra o comando.

Opcionalmente, é possível utilizar a cláusula ***break*** para provocar o encerramento prematuro do laço. Também é possível utilizar a cláusula ***continue*** para provocar o encerramento da iteração atual do laço e o início de uma nova iteração.

6.1.6.4. *While*

Outro comando de repetição da linguagem C é o ***while*** (enquanto). Ele é utilizado para repetir um comando ou bloco de comandos enquanto uma condição for verdadeira. A diferença entre o ***while*** e o ***for*** é que no ***for***, há ciclos de inicialização, teste condicional e incremento, enquanto no ***while*** somente temos o teste condicional.

A sintaxe do ***while*** é:

```
while (condição)
{
      comando1;
      comando2;
      ...
}
```

O funcionamento do comando é bastante simples: primeiramente a condição é avaliada. Caso seja verdadeira, o comando ou bloco de comandos é executado e em seguida a condição é testada novamente. Caso a condição seja falsa, o comando é encerrado. Lembre-se de que cada iteração se inicia com o teste da condição.

Assim como no ***for***, também é possível utilizar a cláusula ***break*** para encerrar prematuramente um laço ***while*** sem necessitar que a condição seja verdadeira, ou a cláusula ***continue*** para provocar o encerramento da iteração atual e início da seguinte.

6.1.6.5. *Do*

O último tipo de comando de repetição é o ***do***. Ele é bastante parecido com o ***while***, mas ao contrário deste, que testa a condição no início de cada iteração, ***do*** somente testa a condição no final de cada iteração. Vejamos a sua sintaxe:

```
do
{
     comando1;
     comando2;
     ...
} while (condição);
```

O seu funcionamento também é bastante simples: o comando ou bloco de comandos é executado e em seguida a condição é verificada. Caso seja verdadeira, uma nova iteração tem início; caso seja falsa, o comando é encerrado.

Aqui também as cláusulas ***break*** e ***continue*** podem ser utilizadas para encerrar o comando ou a iteração prematuramente.

6.1.7. Funções

Para encerrarmos esta breve revisão, vejamos alguns tópicos sobre as funções na linguagem C.

Uma função é um conjunto de comandos que executam uma determinada operação no programa. Normalmente, quando se necessita repetir um mesmo trecho de código diversas vezes em um programa, utiliza-se uma função para guardar esse conjunto de comandos e a cada vez que for necessária a sua execução, basta realizar uma chamada à função.

As funções podem ser equiparadas as sub-rotinas ***assembly***.

Em C, uma função é declarada utilizando a seguinte sintaxe:

```
tipo_de_retorno nome_da_função (lista de parâmetros)
{
    lista_de_comandos;
}
```

Uma função pode retornar qualquer tipo de dado válido em C, e também pode receber qualquer dos tipos de dados válidos em C (com exceção de uniões, conforme já foi comentado anteriormente).

Além disso, uma função C pode receber seus parâmetros de duas formas distintas:

Na chamada por valor, o valor do parâmetro da função é passado como argumento para ela.

Na chamada por referência, o endereço da variável que contém o parâmetro é passado como argumento para ela.

É possível utilizar cada um ou ambos os modelos em uma mesma função, mas o programador deve sempre levar em conta a forma como a função foi declarada, pois a chamada da função deve utilizar a mesma forma da declaração dela.

A utilidade da passagem por referência é permitir que a função altere o valor dos parâmetros passados a ela, o que não é possível na chamada por valor.

Normalmente, C utiliza a chamada por valor, com exceção das matrizes, em que é passado apenas o endereço do primeiro elemento dela. Para utilizar a chamada por referência, é necessária a utilização de ponteiros.

Vejamos um exemplo de declaração de uma função que efetua a média entre dois valores e retorna esse valor para o chamador:

```
unsigned int calcula_media ( unsigned int A, unsigned int B)
{
      return ((A+B) / 2);
}
```

Repare que a função foi declarada como ***unsigned int***, e isso significa que ela vai retornar um valor inteiro de 16 *bits* sem sinal.

O único comando no corpo da função (***return***) é um comando especial que encerra a função e provoca o retorno do programa ao ponto de onde a mesma foi chamada. Neste caso, além de retornar, a função vai devolver o resultado da média.

A função também seria encerrada, ocorrendo o retorno ao ponto de chamada, caso ela atingisse o seu final (fim do bloco de comandos da função, delimitado por "}").

A mesma função pode ser declarada da seguinte forma:

```
unsigned int calcula_media ( unsigned int A, unsigned int B)
{
      unsigned int media;
      media = (A+B) / 2;
      return (media);
}
```

Para utilizar a função "calcula_media", basta incluir uma linha de código no programa:

```
nota = calcula_media (nota1, nota2);
```

ou ainda,

```
nota = calcula_media (10, 8);
```

Para realizar a chamada por referência, é necessário declarar os parâmetros desejados da função como ponteiros. Além disso, a chamada também deve ser realizada utilizando ponteiros.

Para exemplificar isso, vejamos a declaração de uma função para verificar se o parâmetro recebido é maior que 50 e caso seja, subtrair este valor do parâmetro, devolvendo-o alterado:

```
void verifica (unsigned char *va)
{
      if (*va > 50) *va -= 50;
}
```

Observe que a função "verifica" foi declarada como ***void***, pois não retorna valores, além disso, o seu único parâmetro formal é uma variável ponteiro chamada "va", que aponta para um dado do tipo ***char***.

Todas as referências a essa variável feitas dentro da função devem respeitar a sua declaração.

Vejamos um pequeno exemplo de utilização da função "verifica":

```
void main (void)
{
      unsigned char teste;
      teste = 10;
      verifica (&teste); // a variável teste continua igual a 10
      teste = 52;
      verifica (&teste); // a variável teste é igual a 2 após a chamada
}
```

6.2. O Compilador IAR

O compilador C incluso no pacote *Embedded Workbench* é extremamente poderoso e eficiente, compatível com a norma ANSI e inclui suporte para a linguagem C orientada a objetos (C++). Nos próximos tópicos vamos estudar brevemente alguns detalhes sobre a sua arquitetura e funcionamento.

6.2.1. Tipos e Organização dos Dados

O compilador IAR suporta todos os tipos básicos de dados definidos pela norma ANSI para a linguagem C, mais os seguintes tipos adicionais:

Tipo	Faixa de valores	Tamanho em *Bytes*
bool	0 a 1	1
long long	-2^{63} a $2^{63}-1$	8
unsigned long long	0 a $2^{64}-1$	8

Tabela 6-7

O tipo **bool** pertence à linguagem C++, mas é permitido utilizá-lo nos programas em C, desde que as extensões de linguagem estejam ativadas, seja utilizada a biblioteca DLIB e incluído o arquivo "stdbool.h". Mais adiante veremos em maiores detalhes a utilização das bibliotecas CLIB e DLIB. Lembre-se de que uma variável do tipo **bool** ocupa um *byte* de memória para o seu armazenamento.

O tipo inteiro de 64 *bits* (**long long**) está disponível, mas apenas quando utilizada a biblioteca DLIB. Além disso, uma variável desse tipo não pode ser utilizada dentro de um comando **switch**.

Vale lembrar que todas as variáveis inteiras sinalizadas (**signed**) são armazenadas no formato complemento de dois.

As variáveis reais (**float** e **double**) são armazenadas seguindo a recomendação IEE754:

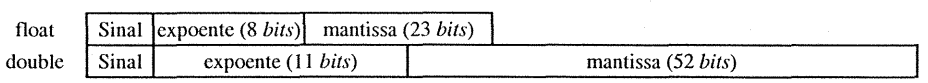


Figura 6-1

Para calcular o valor de um número **float**, utilizamos a seguinte fórmula: $-1^{sinal} * 2^{(expoente-127)} * 1.mantissa$.

Já o valor de um número **double** pode ser calculado pela seguinte fórmula: $-1^{sinal} * 2^{(expoente-1023)} * 1.mantissa$.

6.2.1.1. Uniões, Estruturas e Campos de *Bit*

A alocação de memória para as uniões e estruturas é feita, por padrão, seguindo o alinhamento de 16 *bits* da memória.

No caso de estruturas, o compilador aloca memória do primeiro elemento para o último, preenchendo os eventuais espaços necessários para manutenção do alinhamento da memória.

Assim, uma estrutura declarada como em seguida:

```
struct
{
  char va;              // um byte
  unsigned int vb;      // dois bytes
  char vc;              // um byte
  unsigned long int vd; // quatro bytes
} st;
```

Vai ocupar 10 *bytes* em vez dos esperados 8, porque para manter o alinhamento, o compilador vai inserir um *byte* após o campo "va" e outro após o campo "vc", conforme podemos observar na figura seguinte (os *bytes* de alinhamento são representados por X):

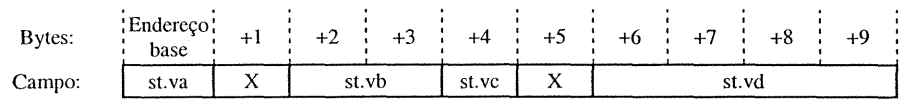


Figura 6-2

Opcionalmente, é possível utilizar a diretiva **#pragma pack(x)** para forçar um determinado alinhamento, em que x indica o número de *bytes* de alinhamento. Assim, a mesma estrutura anterior, se declarada utilizando esta diretiva:

```
#pragma pack(1)
struct
{
  char va;              // um byte
  unsigned int vb;      // dois bytes
  char vc;              // um byte
  unsigned long int vd; // quatro bytes
} st;
```

Seria armazenada utilizando um alinhamento de um *byte* em vez dos dois padrão, fazendo com que a estrutura seja armazenada como segue:

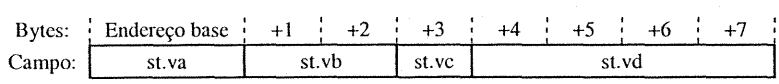


Figura 6-3

Observe que a estrutura ocupa apenas 8 *bytes*, no entanto o campo "vb" está armazenado em um endereço ímpar de memória, o que pode causar uma redução na velocidade de processamento, uma vez que os microcontroladores MSP430 esperam que as variáveis de 16 *bits* estejam sempre localizadas em endereços pares da memória. Isso obriga o compilador a gerar código extra para a manipulação do campo, ou em alguns casos, a variável simplesmente é lida incorretamente. Por isso a mudança de alinhamento deve ser muito bem planejada e testada pelo programador.

No caso de estruturas que utilizam campos de bit, a alocação de memória é feita em grupos de *bytes* e a maior largura de um único campo é de 32 *bits* (64 se utilizado o tipo **long long**).

Normalmente, o primeiro elemento da estrutura é posicionado no *bit* menos significativo do primeiro *byte* da estrutura e os *bits* seguintes são preenchidos seguindo a ordem da estrutura (de

cima para baixo) até o último elemento. A diretiva **#pragma bitfields=reversed** permite inverter a ordem de alocação, do *bit* mais significativo para o menos significativo.

Uma estrutura declarada como segue:

```
struct
{
   char a : 1;
   char b : 1;
   char c : 6;
} st;
```

Ocupa um *byte* de memória preenchido conforme a figura 6.4.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
st.c						st.b	st.a

Figura 6-4

Vejamos o resultado de uma atribuição a um elemento da estrutura:

```
st.a = 1;
```

A atribuição vai gerar o seguinte código:

```
bis.b    #1,&st
```

Utilizando a diretiva **#pragma bitfields=reversed**, a mesma estrutura será alocada na ordem inversa na memória, fazendo com que a eficiência do código utilizado para manipular o campo "três" seja ainda maior, pois o campo está posicionado nos *bits* menos significativos do *byte* de memória.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
um	dois	tres					

Figura 6-5

No caso de estruturas cuja soma dos comprimentos dos campos de *bit* seja superior a 8, o compilador utiliza mais *bytes* para alocar o espaço para os campos restantes. A estratégia para alocação de espaço para os campos de *bits* é simples: por padrão, o compilador aloca espaço iniciando pela parte menos significativa da memória (ou mais significativa, dependendo da opção selecionada) e posiciona os campos de forma a preencher seqüencialmente as posições de memória alocadas para a estrutura.

Para exemplificar o que foi dito anteriormente, vamos apresentar uma estrutura com campos de *bit* um pouco mais complexa e que ocupa 20 *bits*:

```
struct
{
   char a   : 3;
   char b   : 3;
   char c   : 3;
   char d   : 2;
   char e   : 4;
   char f   : 5;
} st2;
```

Se a estrutura apresentada for compilada utilizando o alinhamento-padrão (iniciando do *bit* menos significativo da primeira posição da memória), teremos a seguinte distribuição:

endereço base								endereço + 1								endereço + 2								endereço + 3							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
		st2.b			st2.a						st2.d		st2.c							st2.e							st2.f				

Figura 6-6

Repare na existência de diversos espaços em branco, originados devido à estratégia de alocação do compilador.

Finalmente, devemos levar em consideração o tipo de dado utilizado para declarar cada campo de *bit*: campos do tipo **char** estão limitados a 8 *bits*, **int** a 16 *bits* e assim por diante.

6.2.2. Convenções de Chamada e de Retorno de Funções

O compilador IAR adota as convenções de chamada e retorno de funções padronizadas pelo comitê ANSI.

A chamada de funções utiliza os registradores R12 a R15 e eventualmente R8 a R11. Além destes o compilador pode utilizar também a pilha para a passagem de parâmetros.

- Um parâmetro do tipo **char**, **int** ou um ponteiro de memória são passados utilizando apenas um registrador da CPU, iniciando pelo R12.
- Um parâmetro **long int** ou **float** utiliza um par de registradores, R13:R12 ou R15:R14 (parte mais significativa em R13 ou R15).
- Um parâmetro **long long** ou **double** utiliza quatro registradores: R15:R12 ou R11:R8 (parte mais significativa em R15 ou R11).
- Os demais parâmetros são passados nos registradores (R12 a R15) restantes. Caso não existam registradores suficientes, o compilador utiliza a pilha.
- Uma exceção às regras anteriores são os seguintes parâmetros, que sempre são passados por meio da pilha:
  - Estruturas, uniões e classes;
  - Número variável de parâmetros (como na função **printf**).

Um caso especial são as funções que retornam estruturas ou matrizes. Neste caso, o registrador R12 é utilizado para armazenar o endereço (ponteiro) da variável.

Quando os parâmetros são passados pela pilha, ela é alocada da seguinte forma:

Endereço	Conteúdo da pilha
n	Endereço de retorno da função
n–2	primeiro parâmetro
n–4	segundo parâmetro
n–6	terceiro parâmetro
n–8	enésimo parâmetro
n–10	Topo anterior da pilha (antes da chamada da função)

Tabela 6-8

Vale lembrar que a pilha somente é utilizada para os parâmetros que não puderem ser passados pelos registradores da CPU, ou nos casos especiais já citados.

O retorno de valores da função é feito sempre por intermédio dos registradores da CPU. Neste caso, o compilador utiliza o registrador R12 para valores de 8/16 *bits* (**char**, **int** e ponteiros), R13:12 para valores de 32 *bits* (**long int** e **float**) e R15:R12 ou R11:R8 para valores de 64 *bits* (**long long** ou **double**).

6.2.3. Funções Intrínsecas

O compilador dispõe de diversas funções intrínsecas que podem ser utilizadas para acesso direto a alguns elementos da CPU, instruções especiais, modos de baixa potência, controle de interrupções, etc.

O uso dessas funções está condicionado à inclusão do arquivo "intrinsics.h":

__bcd_add_short - adiciona dois valores de 16 *bits* em formato BCD (decimal codificado em binário). Por exemplo:

```
unsigned int a,b,c;
b = 0x09;
c = 0x02;
a = __bcd_add_short(b,c);       // a = 0x0011
```

__bcd_add_long - adiciona dois valores de 32 *bits* em formato BCD (decimal codificado em binário). Por exemplo:

```
unsigned long int a,b,c;
b = 0x0105;
c = 0x0205;
a = __bcd_add_long(b,c);        // a = 0x00000310
```

__bcd_add_long_long - adiciona dois valores de 64 *bits* em formato BCD (decimal codificado em binário). Por exemplo:

```
unsigned long long a,b,c;
b = 0x0105;
c = 0x0205;
a = __bcd_add_long_long(b,c);  // a = 0x0000000000000310
```

__bic_SR_register - apaga os *bits* especificados do registrador SR. O parâmetro fornecido é a máscara de *bits* a serem apagados. Por exemplo:

```
__bic_SR_register(1);   // apaga o bit C (carry)
```

__bic_SR_register_on_exit - apaga os *bits* especificados do registrador SR salvos na pilha, antes do retorno de uma função de tratamento de interrupção ou monitor. Essa função somente pode ser utilizada dentro de funções de interrupção ou monitor. O parâmetro fornecido é a máscara de *bits* a serem apagados. Por exemplo:

```
__bic_SR_register_on_exit(GIE);// apaga o bit GIE salvo na pilha
```

__bis_SR_register - seta os *bits* especificados do registrador SR. O parâmetro fornecido é a máscara de *bits* a serem setados. Por exemplo:

```
__bic_SR_register(1);   // seta o bit C (carry)
```

__bis_SR_register_on_exit - seta os *bits* especificados do registrador SR salvos na pilha, antes do retorno de uma função de tratamento de interrupção ou monitor. Essa função somente pode ser utilizada dentro de funções de interrupção ou monitor. O parâmetro fornecido é a máscara de *bits* a serem setados. Por exemplo:

```
__bis_SR_register_on_exit(GIE);// seta o bit GIE salvo na pilha
```

__disable_interrupt - apaga o *bit* GIE no SR, desabilitando as interrupções. Por exemplo:

```
__disable_interrupt();
```

__enable_interrupt - seta o *bit* GIE no SR, habilitando as interrupções. Por exemplo:

```
__enable_interrupt();
```

__even_in_range - instrui o compilador para gerar código otimizado, considerando que o primeiro argumento é um endereço par e que o segundo é a faixa de endereços válida. Essa função é especialmente útil para o tratamento dos registradores geradores de vetor de interrupção, como TAIV, TBIV, I2CIV, ADC12IV, etc. Por exemplo:

```
#pragma vector=TIMERA1_VECTOR
__interrupt void trata_timer_A (void)
{
  switch (__even_in_range(TAIV, 10))
  {
     case 2:
               // comandos
               break;
     case 4:
               // comandos
               break;
     case 10:
               // comandos
               break;
  }
}
```

__get_interrupt_state - retorna o estado atual do *bit* GIE. O tipo **istate** pode ser utilizado para declarar a variável que irá receber o resultado da função. Por exemplo:

```
istate_t estado_anterior = __get_interrupt_state(void);
__enable_interrupt();
...
__set_interrupt_state(estado_anterior);
```

__get_R4_register - retorna o conteúdo do registrador R4 (somente quando ele está travado). Por exemplo:

```
unsigned int R4_reg;
R4_reg = __get_R4_register ();
```

__get_R5_register - retorna o conteúdo do registrador R5 (somente quando o mesmo está travado). Por exemplo:

```
unsigned int R5_reg;
R5_reg = __get_R5_register();
```

__get_SP_register - retorna o conteúdo do registrador SP. Por exemplo:

```
unsigned int SP_reg;
SP_reg = __get_SP_register();
```

__get_SR_register - retorna o conteúdo do registrador SR. Por exemplo:

```
unsigned int SR_reg;
SR_reg = __get_SR_register();
```

__get_SR_register_on_exit - retorna o conteúdo do registrador SR salvo na pilha e que será restaurado após o retorno da interrupção ou função monitor. Essa função somente pode ser utilizada dentro de funções de interrupção ou monitor. Por exemplo:

```
unsigned int SR_reg;
SR_reg = __get_SR_register_on_exit();
```

__low_power_mode_n - configura o registrador SR para entrada no modo de baixa potência "n". Por exemplo:

```
__low_power_mode_1();   // entrada no modo LPM1
__low_power_mode_4();   // entrada no modo LPM4
```

__low_power_mode_off_on_exit - configura o registrador SR salvo na pilha para que após a saída da interrupção a CPU permaneça no modo normal (saia do modo de baixa potência). Essa função somente pode ser utilizada dentro de funções de interrupção ou monitor. Por exemplo:

```
__low_power_mode_off_on_exit();
```

__no_operation - insere uma instrução NOP. Por exemplo:

```
__no_operation();
```

__op_code - insere um *opcode* de uma instrução *assembly*. Por exemplo:

```
__op_code(0x4303);
```

__segment_begin - retorna o endereço do primeiro *byte* do segmento especificado.

__segment_end - retorna o endereço do primeiro *byte* após o segmento especificado.

__set_interrupt_state - configura o *bit* GIE de acordo com o estado especificado. Por exemplo:

```
istate_t estado_anterior = __get_interrupt_state(void);
__enable_interrupt();
...
__set_interrupt_state(estado_anterior);
```

__set_R4_register - seta o conteúdo do registrador R4 (somente quando ele está travado). Por exemplo:

```
__set_R4_register (0x10);
```

__set_R4_register - seta o conteúdo do registrador R5 (somente quando ele está travado). Por exemplo:

```
__set_R5_register (0x20);
```

__set_SP_register - seta o conteúdo do registrador SP. Por exemplo:

```
__set_SP_register (0x1000);
```

6.2.4. Diretivas e Extensões da Linguagem

Além das palavras reservadas da linguagem C padrão, o compilador IAR suporta ainda as seguintes:

@ - permite especificar o endereço de uma variável. Somente pode ser especificado o endereço de uma variável declarada como **__no_init** ou como **const**.

asm ou **__asm** - insere uma instrução *assembly* ou diretiva do montador diretamente no ponto em que foi declarada. Por exemplo:

```
asm ("MOV R12, R13");
```

__data16 - armazena a variável ou constante no segmento *data16* da memória. O tamanho máximo do objeto é 32*Kbytes* e o tamanho máximo da memória é 64*Kbytes*. Por exemplo:

```
__data16 unsigned int variavel;
```

__interrupt - informa ao compilador que a função seguinte é de tratamento de interrupção. Funções desse tipo devem ser declaradas como **void**. Por exemplo:

```
#pragma vector=WDT_VECTOR
__interrupt void trata_wdt (void)
{
}
```

__monitor - informa ao compilador que na função seguinte as interrupções devem permanecer desativadas. O compilador desativa e ativa automaticamente o GIE na entrada/saída da função. Por exemplo:

```
__monitor void funcao_prioritaria (void)
{
}
```

__noinit - suprime a inicialização da variável declarada. Por exemplo:

```
__no_init unsigned int variavel;
```

__noreturn - informa ao compilador que a função seguinte não irá retornar. Isso gera um código mais eficiente, mas a sua utilidade é discutível. Por exemplo:

```
__noreturn void funcao_x (void)
{
}
```

__raw - informa ao compilador para não gerar código para preservar os registradores na entrada e saída da função de interrupção declarada. Por exemplo:

```
__raw __interrupt void trata_interrupcao (void)
{
}
```

__regvar - informa ao compilador que a variável declarada deve ser armazenada em um registrador da CPU (R4 ou R5). Para utilizá-la, é necessário reservar esses registradores pela opção R4/R5 *utilization* na janela de opções do projeto, categoria C/C++ *compiler*, aba *Code*. Por exemplo:

```
__regvar __no_init unsigned int var_reg @ __R4;
```

Variáveis **__regvar** somente podem ser do tipo **char**, **int**, **long int**, **float**, ponteiro, estrutura com um campo ou uma união destes tipos. Não é possível obter o endereço de uma variável **__regvar** pelo operador &.

__root - informa ao compilador que a função ou variável declarada deve ser incluída, mesmo que não utilizada pelo programa. Por exemplo:

```
__root unsigned char temp;
```

__task - informa ao compilador para não restaurar os registradores da CPU utilizados na função. Esta opção pode ser útil em funções utilizadas em aplicações de tempo real. Por exemplo:

```
__task void funcao_x (void)
{
}
```

__word16 - armazena a variável ou constante no segmento *word16* da memória. O tamanho máximo do objeto é 32*Kbytes* e o tamanho máximo da memória é 128*Kbytes*. Por exemplo:

```
__word16 unsigned int variavel;
```

Atenção: não é permitido o acesso a *bytes* no segmento *word*16 !

Além das palavras anteriores, estão disponíveis também algumas diretivas de compilação que podem ser utilizadas com **#pragma**:

bis_nmi_ie1 - informa ao compilador para inserir uma instrução para ativação de um ou mais bits de habilitação de interrupções NMI ao término da função. Essa opção garante que as interrupções NMI somente sejam re-ativadas imediatamente antes do retorno da interrupção. Por exemplo:

```
#pragma bis_nmi_ie1 = OFIE
__interrupt void trata_NMI (void)
{
  // imediatamente antes do retorno da função
  // o bit OFIE é automaticamente ativado
}
```

bitfields - seleciona o tipo de alinhamento dos campos de *bit*: lsb para msb (*default*) ou msb para lsb (*reversed*). Para maiores detalhes veja o tópico 6.2.1.1 na página 338.

inline - solicita ao compilador que a função seguinte tenha o seu código inserido no local da sua chamada, acelerando a execução, mas aumentando o tamanho do programa. O compilador decide pela inserção da função ou de uma chamada para ela. Por exemplo:

```
#pragma inline
void funcao_inline (void)
{
}
```

Opcionalmente é possível utilizar **#pragma inline = forced** para forçar o compilador a somente utilizar a função na forma *inline*.

__location - informa ao compilador para armazenar a variável seguinte no endereço ou segmento especificado. O nome do segmento deve aparecer entre aspas. Por exemplo:

```
#pragma location = 0x22E
unsigned char PORT1;
```

__pack - altera a forma de alinhamento na memória de estruturas e uniões. Para maiores detalhes veja o tópico 6.2.1.1 na página 338.

__vector - informa ao compilador que o vetor de interrupção especificado deve apontar para a próxima função. Por exemplo:

```
#pragma vector=WDT_VECTOR
__interrupt void trata_wdt (void)
{
}
```

6.2.5. Bibliotecas C

O compilador IAR prevê o uso de dois tipos de biblioteca compilada de funções: CLIB e DLIB. Essas bibliotecas incluem diversas funções-padrão úteis ao programador e que podem ser utilizadas no programa pela simples inclusão de um arquivo de cabeçalho do tipo "nome_da_biblioteca.h".

A diferença básica entre as duas está na complexidade e tamanho, como veremos a seguir.

6.2.5.1. CLIB

A biblioteca CLIB possui um caráter mais básico, sendo parcialmente compatível com o padrão ANSI, por isso ela também é menor, reduzindo a quantidade de memória ocupada.

Na biblioteca CLIB estão disponíveis as seguintes bibliotecas-padrão:

Nome	Função
assert.h	Depuração através de mensagens no dispositivo padrão de saída de erros (STDERR)
ctype.h	Manipulação de caracteres
errno.h	Valores de retorno de erros
float.h	Limites e tamanho dos tipos ponto flutuante
iccbutl.h	Rotinas de baixo nível utilizadas em funções intrínsecas
limits.h	Limites e tamanho dos tipos inteiros
math.h	Funções matemáticas
setjmp.h	Desvios não locais
stdarg.h	Utiliza na criação de funções com número variável de argumentos
stddef.h	Definições comuns
stdio.h	Entrada/saída
stdlib.h	Utilidade geral
string.h	Manipulação de strings

Tabela 6-9

A seleção da biblioteca CLIB pode ser feita na janela de opções do projeto, categoria *General Options*, aba *Library Configuration*. É possível utilizar a biblioteca CLIB previamente compilada ou a opção *Custom* CLIB que permite recompilar a biblioteca CLIB de acordo com as necessidades do programador.

Existem quatro arquivos-fonte em *assembly*, classificados de acordo com o tamanho do tipo **double** suportado e se o código gerado é ou não independente de posição. Esses arquivos são automaticamente selecionados pelo compilador a partir das opções feitas na janela de opções do projeto.

Arquivo	Características	
	Tamanho do double	Código independente de posição
cl430f.r43	32	Não
cl430fp.r43	32	Sim
cl430d.r43	64	Não
cl430dp.r43	64	Sim

Tabela 6-10

6.2.5.2. DLIB

A biblioteca DLIB pode ser encontrada em duas versões diferentes: DLIB normal e DLIB completa. Ambas são compatíveis com o padrão ANSI, no entanto a DLIB normal não implementa as funcionalidades de suporte à conversão de localidades, suporte a *multibytes* nas funções **printf** e **scanf** e não inclui suporte à conversão de *strings* para números **float** hexadecimais na função **strtod**. Na biblioteca DLIB estão disponíveis as seguintes bibliotecas C padrão:

Nome	Função
assert.h	Depuração através de mensagens no dispositivo padrão de saída de erros (STDERR)
ctype.h	Manipulação de caracteres
errno.h	Valores de retorno de erros
iso646.h	Arquivo de cabeçalho compatível com a norma ISO646
limits.h	Limites e tamanho dos tipos inteiros
locale.h	Adaptações para outros idiomas e localidades
math.h	Funções matemáticas
setjmp.h	Desvios não locais
signal.h	Controle de condições de exceção
stdarg.h	Utilizada na criação de funções com número variável de argumentos
stdbool.h	Suporte para tipos booleanos em C
stddef.h	Definições de tipos de dados úteis e macros
stdio.h	Entrada/saída
stdlib.h	Utilidade geral
string.h	Manipulação de strings
time.h	Conversões entre diversos tipos e formatos de representação de data e horário
wchar.h	Suporte para caracteres de largura estendida
wctype.h	Classificação de caracteres de largura estendida

Tabela 6-11

A seleção da biblioteca DLIB pode ser feita na janela de opções do projeto, categoria *General Options*, aba *Library Configuration*. É possível utilizar uma das bibliotecas DLIB previamente compiladas (normal ou *full* (completa)) ou a opção *Custom* DLIB que permite recompilar a biblioteca DLIB de acordo com as necessidades do programador.

Existem oito arquivos-fonte em *assembly*, classificados de acordo com o tamanho do tipo **double** suportado, o tamanho da biblioteca (normal ou completa) e se o código gerado é ou não independente de posição. Esses arquivos são automaticamente selecionados pelo compilador a partir das opções feitas na janela de opções do projeto.

Arquivo	Características		
	Tamanho do double	Tamanho da biblioteca	Código independente de posição
dl430fn.r43	32	normal	Não
dl430fnp.r43	32	normal	Sim
dl430ff.r43	32	completa	Não
dl430ffp.r43	32	completa	Sim
dl430dn.r43	64	normal	Não
dl430dnp.r43	64	normal	Sim
dl430df.r43	64	completa	Não
dl430dfp.r43	64	completa	Sim

Tabela 6-12

A personalização das bibliotecas está detalhada na documentação eletrônica que acompanha o compilador.

6.2.6. Embutindo Código Assembly

O compilador IAR inclui um ótimo suporte à utilização da linguagem *assembly* dentro de programas escritos em C.

O suporte pode ser classificado em duas categorias básicas:

- Instruções *assembly* embutidas diretamente no código C, utilizando a palavra reservada **asm**;
- Funções externas escritas em *assembly* e ligadas ao programa em C pelo ligador.

No caso do uso de instruções *assembly* dentro de uma função C, devemos ter alguns cuidados e precauções:

1. As variáveis C podem ser acessadas diretamente por uma instrução *assembly*, utilizando o seu próprio nome. No entanto, lembre-se de que o compilador pode adotar algumas estratégias de otimização que podem impedir o acesso a elas. Isso ocorre com variáveis locais e com os parâmetros recebidos dentro de uma função.
2. Não é possível utilizar *labels* (etiquetas).
3. Algumas diretivas do montador não estão disponíveis.
4. O compilador não avalia, para fins de otimização, o código *assembly* introduzido por **asm**. Isso significa que o programador deve verificar muito bem o código circundante e o código gerado, a fim de certificar-se de que o efeito desejado foi produzido.

De forma geral, não é recomendada a utilização de código *assembly* embutido, a não ser em casos de extrema necessidade.

No segundo caso, o uso de funções externas escritas em *assembly*, é necessário prototipá-las utilizando a palavra reservada **extern** e incluir o arquivo *assembly* no projeto.

A função externa deve obedecer às regras de passagem e retorno de parâmetros apresentadas no tópico 6.2.2.

6.2.7. Produzindo Código C Eficiente

A produção de um código C eficiente é crucial quando se trata da programação de dispositivos tão limitados como os microcontroladores.

Quando se fala em programar um microcontrolador dotado de 1Kb de memória FLASH e 128 *bytes* de RAM, a maioria dos programadores pode, a princípio, descartar uma linguagem de alto nível como C, no entanto o uso de boas técnicas de programação pode viabilizar esse cenário.

Alguns bons procedimentos de programação em C:

1. Procure utilizar tipos de dados mais facilmente manipuláveis pelo microcontrolador. No caso dos MSPs, os tipos de 8 e 16 *bits* devem ter preferência.
2. Evite operações com números sinalizados, as quais tendem a ser mais lentas que as operações com números sem sinal.
3. Procure concetrar as operações com *bits* nos quatro *bits* LSB (constantes 0, 1, 2, 4 ou 8), pois isso permite que o compilador utilize os geradores de constantes nas operações, reduzindo o tamanho do código e aumentando a sua velocidade.
4. Evite utilizar variáveis ***float*** e ***double***: o uso de variáveis do tipo ponto flutuante deve ser ponderado e avaliado, pois na grande maioria dos casos, ele não é necessário.

Prefira utilizar variáveis inteiras maiores e multiplique os valores por uma constante, de acordo com a precisão desejada. Por exemplo: supondo que será feita a medição de uma tensão de 0 a 10V, com precisão de três casas decimais. Isso significa que a menor tensão será 0,000V e a maior 10,000V. Como queremos três casas decimais, multiplicamos o valor por 1.000, assim uma tensão de 0,001 Volts será representada pelo valor 1 e uma tensão de 10 Volts será representada pelo valor 10.000. Isto permite que utilizemos uma variável inteira de 16 *bits* em vez de um ***float*** de 32 *bits*.

5. A utilização de variáveis locais deve ser privilegiada, em detrimento das variáveis globais. As variáveis locais somente ocupam memória durante a execução da função em que foram declaradas, liberando a memória para a aplicação durante o resto da execução do programa.
6. A utilização de funções *inline* algumas vezes pode produzir um código mais eficiente, já que elas economizam as operações que envolvem a passagem e retorno de valores.
7. Evite o uso de funções recursivas (que chamam a si mesmas). Esse tipo de função pode consumir rapidamente o espaço de memória da pilha.
8. Evite a passagem de parâmetros muito grandes, tais como estruturas, para uma função. Isso pode consumir muita pilha de memória. Em vez disso, utilize ponteiros para referenciar a variável desejada.

7

Exemplos de Aplicação

Neste capítulo, apresentamos alguns projetos e programas que demonstram as capacidades dos microcontroladores MSP430. Os exemplos podem ser baixados diretamente do site da Editora.

7.1. Controlando o DCO

No primeiro exemplo, vamos demonstrar um programa muito simples que pode ser utilizado para efetuar o ajuste do oscilador interno (DCO) nos *chips* da família 1xx (e 2xx com pequenas modificações).

O princípio básico é o seguinte: uma aplicação de baixa potência tende a necessitar trabalhar com freqüências de *clock* baixas, na ordem de dezenas ou centenas de KHz, de forma a diminuir o consumo de corrente.

No entanto, muitas dessas aplicações podem precisar trabalhar com freqüências de *clock* mais elevadas, mesmo que por breves instantes, seja para aumentar a velocidade de execução do programa em rotinas mais complexas, ou para permitir que um periférico utilize uma freqüência de *clock* mais elevada (por exemplo, uma USART durante uma transmissão, um *timer* durante uma captura, etc.).

O projetista tem duas opções:

1. Utilizar um segundo cristal, de alta freqüência, que somente seria ativado quando necessário. Esta alternativa garante uma elevada precisão do *clock*, mas agrega problemas como aumento do consumo, tempo de estabilização do oscilador, custo, ruído eletromagnético, etc.
2. Utilizar o oscilador interno DCO que apresenta uma partida extremamente rápida e um baixo consumo de corrente, mas tem a desvantagem de ser impreciso.

O programa apresentado a seguir pode ser utilizado para auxiliar na configuração do DCO. Ele compara a freqüência de uma fonte conhecida e estável, no caso o oscilador LFXT1 trabalhando a 32768Hz, com a freqüência de operação do DCO. Atuando sobre os registradores de controle do DCO, o programa consegue configurá-lo para trabalhar em uma freqüência muito próxima da configurada pelo programador.

```
//*******************************************************************
// Ajuste do DCO
//*******************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*******************************************************************
// Este programa demonstra o ajuste do DCO a partir do oscilador
```

```
// LFXT1 operando em baixa freqüência (32768Hz)
// Isto permite construir aplicações que necessitem da medição
// permanente e precisa de tempo (utilizando o cristal de 32768Hz)
// ao mesmo tempo em que, quando necessário, a CPU e periféricos
// podem utilizar um clock de alta freqüência, originado pelo DCO
// O ajuste efetuado por este tipo de programa, garante uma grande
// precisão na seleção da freqüência do DCO
// O funcionamento é simples: utilizamos o watchdog operando como um
// timer, a fonte de clock do mesmo é o ACLK, que neste caso é igual
// a 32768Hz. Programando-se o watchdog para gerar uma interrupção a
// cada 64 pulsos do ACLK, teremos uma base de tempo de referência
// para a medição do sinal gerado pelo DCO.
// Utilizamos também o timer A, que será o responsável pela contagem
// dos pulsos de clock do DCO, para isso, configuramos a fonte de
// clock do timer como sendo o SMCLK e configuramos este clock para
// ser originado do DCO.
// Feito isso, a cada interrupção do watchdog, verificamos a contagem
// do TAR.
// Se quisermos trabalhar com uma frequência de 1MHz, a cada inter-
// rupção do watchdog, o TAR deverá ter feito 1953 contagens. Se o
// número de contagens for maior, quer dizer que a freqüência do DCO
// é maior que 1MHz, se for menor, também menor é a freqüência do DCO.
//*********************************************************************

#include <io430x14x.h>
#include <intrinsics.h>

#define FCLOCK 4000000   // freqüência desejada em Hz

#define clock_ok 1

// estrutura utilizada para o controle da função de ajuste de clock
struct sclkadj
{
   char ajustando : 1;
   char err : 1;
} clock_result;   // variável global

// variáveis de ajuste do DCO
unsigned char ajusteMODx, ajusteDCOx, ajusteRSELx;

//*********************************************************************
// aumenta_DCO
//*********************************************************************
// Entrada: void  Saída: unsigned char
//*********************************************************************
// A cada chamada desta função a freqüência do DCO é incrementada em
// um passo. Isto é feito incrementando-se o MODx até atingir 31, ao
// ultrapassar este valor, o DCOx é aumentado até ser maior que 224,
// em seguida, o RSELx é incrementado até 7. Caso a função seja cha-
// mada e necessite incrementar RSELx acima de 7, é retornado o valor
// 0, indicando o erro. Caso o aumento seja bem sucedido, retorna 1
//*********************************************************************
unsigned char aumenta_DCO(void)
{
   unsigned char result = 1;
   // se MODx é menor que 31, incrementa
   if (ajusteMODx<31) ajusteMODx++; else
   {
          // se MODx é maior que 31
          ajusteMODx = 0;        // MODx = 0
          // se DCOx é menor quye 224, aumenta 32
          if (ajusteDCOx<224) ajusteDCOx+=32; else
          {
                 // se DCOx é maior que 224
```

```
                // apaga DCOx
                ajusteDCOx = 0;
                // se RSELx é menor que 7, incrementa
                if (ajusteRSELx<7) ajusteRSELx++;
                        // senão, o resultado é falso
                        else result = 0;
        }
    }
    // configura o DCOCTL de acordo com as variáveis de ajuste
    DCOCTL = ajusteDCOx + ajusteMODx;
    // configura o BCSCTL1 de acordo com a variável de ajuste dos  RSELx
    BCSCTL1 = ajusteRSELx;
    // retorna o resultado
    return (result);
}

void ajusta_clock(void)
{
    unsigned int IE1_temp, TACTL_temp;
    // configura o WDT para o modo temporizador, com clock = ACLK
    // fator de divisão de 64
    WDTCTL = WDTPW + WDTTMSEL + WDTCNTCL + WDTSSEL + WDTIS1 + WDTIS0;
    IFG1_bit.WDTIFG = 0;   // apaga o sinalizador de interrupção do watchdog
    IE1_temp = IE1;        // salva o registrador IE1
    IE1_bit.WDTIE = 1;     // habilita a interrupção do watchdog
    // configura o DCO de acordo com a faixa de freqüência selecionada
    BCSCTL2 = 0;
    ajusteMODx = 0;
    #if FCLOCK<200000
            BCSCTL1 = 0;
            DCOCTL = 0;
            ajusteDCOx = 0;
            ajusteRSELx = 0;
    #elif FCLOCK<500000
            BCSCTL1 = 1;
            DCOCTL = 64;
            ajusteDCOx = 64;
            ajusteRSELx = 1;
    #elif FCLOCK<1000000
            BCSCTL1 = 3;
            DCOCTL = 64;
            ajusteDCOx = 64;
            ajusteRSELx = 3;
    #elif FCLOCK<2000000
            BCSCTL1 = 4;
            DCOCTL = 96;
            ajusteDCOx = 96;
            ajusteRSELx = 4;
    #elif FCLOCK<3000000
            BCSCTL1 = 6;
            DCOCTL = 32;
            ajusteDCOx = 32;
            ajusteRSELx = 6;
    #else
            BCSCTL1 = 7;
            DCOCTL = 32;
            ajusteDCOx = 32;
            ajusteRSELx = 7;
    #endif
    // salva a configuração atual do timer A
    TACTL_temp = TACTL;
    // configura o Timer A:
    // modo 2, fonte de clock = SMCLK
    TACTL = TASSEL_2 + MC_2 + TACLR;
    __enable_interrupt(); // habilita interrupções
```

```
    // apaga indicador de erro
    clock_result.err = 0;
    // ativa indicador de ajuste em andamento;
    clock_result.ajustando = 1;
    while (clock_result.ajustando);      // aguarda o término do ajuste
    // desliga o watchdog
    WDTCTL = WDTPW + WDTHOLD;
    // restaura a configuração do timer A e apaga a contagem
    TACTL = TACTL_temp + TACLR;
    // restaura a configuração de interrupções anterior
    IE1 = IE1_temp;
}

//*********************************************************************
// trata_wdt
//*********************************************************************
// Entrada: void Saída: void
//*********************************************************************
// Função de tratamento de interrupção. A cada 64 pulsos de clock do
// ACLK, o watchdog gera esta interrupção. Ela verifica o número de
// contagens feitas no timer A e caso menor que o valor calculado,
// aumenta a freqüência do DCO em um passo.
// Quando a quantidade de pulsos contadosé igual ou maior ao valor
// programado, a variável indicadora de ajuste é apagada
//*********************************************************************
#pragma vector=WDT_VECTOR
__interrupt void trata_wdt (void)
{
    unsigned int temp;
    temp = TAR;     // armazena temporáriamente a contagem do TAR
    TACTL_bit.TACLR = 1;  // apaga a contagem do TAR
    // se a contagem for menor que o valor programado
    if (temp < FCLOCK/512)
    {
            // se a função de aumento da freqüência do DCO retornar
            // falso, termina o ajuste
            if (!aumenta_DCO()) clock_result.ajustando = 0;
    // se a contagem for maior ou igual ao o valor programado
    // termina o ajuste
    } else clock_result.ajustando = 0;
}

void main( void )
{
    // o loop principal demonstra o funcionamento ...
    // a freqüência do DCO pode ser visualizada com um osciloscópio através
    // do pino P5.4 (nos chips F13x, F14x, F15x e F16x)
    P5SEL_bit.P5SEL_4 = 1;// configura o pino P5.4 para a função alternativa
    P5DIR_bit.P5DIR_4 = 1;// configura o pino P5.4 como saída
    // o pino P5.6 pode ser utilizado para medir o tempo de ajuste da
    // freqüência
    P5DIR_bit.P5DIR_6 = 1;// configura o pino P5.4 como saída
    P5OUT_bit.P5OUT_6 = 1;// seta o pino
    ajusta_clock();       // chama a função de ajuste do DCO
    P5OUT_bit.P5OUT_6 = 0;// desliga o pino
    while(1);
}
```

Exemplo 7-1

Como sugestões de modificações futuras para este exemplo podemos citar: possibilidade de utilizar outras freqüências de referência, além de 32.768Hz, a implementação de uma função na qual se pudesse fornecer como parâmetro a freqüência de *clock* desejada, etc.

7.2. Módulo LCD 16x2 Caracteres

Neste tópico vamos estudar a implementação de algumas funções em linguagem C para comunicação com módulos LCD dotados de controladores KS0066 ou HD44780. Esses controladores possuem 80 *bytes* de memória RAM para os dados do display (DDRAM), 64 *bytes* de RAM para o gerador de caracteres do usuário (CGRAM) e 9.920 *bits* para a ROM do gerador de caracteres.

O módulo utilizado (Intech ITM1602B) apresenta um conector de 16 pinos cuja função encontra-se listada na tabela 7-1.

Pino	Nome	Função
1	V_{SS}	Referência de terra
2	V_{DD}	Alimentação (+5V)
3	V_O	Tensão de contraste
4	RS	Seleção entre modo de comando (0) ou dados (1)
5	R/$\overline{W}$	Seleção entre escrita (0) ou leitura (1)
6	E	Habilitação - em nível "1", o *display* pode receber comandos ou dados ou ainda enviar dados
7	DB0	*Bit* 0 do dado/comando. Não utilizado no modo de 4 *bits*
8	DB1	*Bit* 1 do dado/comando. Não utilizado no modo de 4 *bits*
9	DB2	*Bit* 2 do dado/comando. Não utilizado no modo de 4 *bits*
10	DB3	*Bit* 3 do dado/comando. Não utilizado no modo de 4 *bits*
11	DB4	*Bit* 4 do dado/comando
12	DB5	*Bit* 5 do dado/comando
13	DB6	*Bit* 6 do dado/comando
14	DB7	*Bit* 7 do dado/comando
15	A	Anodo do LED de *backlight*
16	K	Catodo do LED de *backlight*

Tabela 7-1

A DDRAM é organizada de forma que os endereços de 0x00 a 0x27 armazenam os dados da primeira linha e de 0x40 a 0x67, os dados da segunda linha. É claro que desses 40 *bytes* de cada linha, apenas 16 em cada uma são visíveis (num *display* de 16 caracteres). A figura 7-1 apresenta a organização da DDRAM.

Linha	Endereço																																							
1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27
2	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67

Figura 7-1

No modo normal (sem deslocamento da imagem) o primeiro caractere da primeira linha é o do endereço 0x00, o segundo o do endereço 0x01 e assim por diante.

Com relação a CGRAM, trata-se de uma área de memória dedicada à construção de caracteres pelo próprio usuário. Nos 64 *bytes* da CGRAM, é possível a criação de 8 caracteres de 8x5 ou 4 de 10x5. Esses caracteres do usuário podem ser selecionados pelos códigos da primeira coluna da tabela da figura 7-2.

A ROM do gerador de caracteres contém uma tabela de 208 caracteres 8x5 e 32 caracteres 10x5, conforme pode ser visto na figura 7-1 (que representa o conjunto de caracteres da ROM de código A00).



Figura 7-2

Para utilizar um módulo LCD, é necessário conhecer o *hardware* do módulo a ser utilizado. No presente caso, foi utilizado o modelo SSC2E16DLNW-E com duas linhas de 16 caracteres, controlador KS0066, apresentando caracteres pretos sobre fundo verde, proveniente de um *backlight* a LED.

Esses módulos podem trabalhar em duas modalidades distintas de barramento de comunicação: 4 ou 8 *bits*. No modo de 8 *bits*, os dados e comandos são lidos e escritos pelo barramento de 8 *bits* do módulo (pinos DB0 a DB7). No modo de 4 *bits*, utilizam-se apenas os pinos DB4 a DB7 e a informação é enviada em duas etapas. Nos exemplos apresentados neste livro, utilizaremos o modo 4 *bits*, por uma questão de economia de pinos do microcontrolador.

Para utilizar o módulo no modo de 4 *bits* é necessário utilizar uma seqüência de inicialização determinada pelo fabricante, conforme veremos mais adiante.

No entanto, antes de mais nada, é necessário conhecer os tipos de comandos disponíveis no módulo e a sua forma de utilização.

A tabela 7-2 apresenta todos os comandos possíveis para os módulos dotados do controlador KS0066 ou HD44780.

Comando	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Descrição	Duração
Apaga *Display*	0	0	0	0	0	0	0	0	0	1	Apaga o *display* e retorna o cursor para a primeira posição da primeira linha.	-
Retorno	0	0	0	0	0	0	0	0	1	x	Retorna o cursor para a primeira posição da primeira linha e retorna a mensagem ao formato original caso estivesse deslocada.	1,52ms
Configura modo	0	0	0	0	0	0	0	1	I/D	S	I/D - Configura o sentido de deslocamento do *display*: 1 - movimenta o cursor à direita para cada caractere escrito; 0 - movimenta o cursor à esquerda para cada caractere escrito; S - ativa (1) ou desativa (0) o deslocamento da mensagem a cada leitura da DDRAM.	37µs
Controle Liga/Desliga do *display*	0	0	0	0	0	0	1	D	C	B	Liga (D=1) ou desliga (D=0) o *display*, ativa (C=1) ou desativa (C=0) o cursor e ativa (B=1) ou desativa (B=0) o cursor piscante.	37µs
Deslocamento	0	0	0	0	0	1	S/C	R/L	x	x	S/C - seleciona entre deslocamento do cursor (0) e cursor + mensagem (1) R/L - Sentido do deslocamento: direita (1) ou esquerda (0).	37µs
Configuração do *display*	0	0	0	0	1	DL	N	F	x	x	DL - número de *bits* do barramento: 1 - 8 *bits* ou 0 - 4 *bits*. N - número de linhas: 1 - 2 linhas, 0 - 1 linha. F - tamanho dos caracteres: 1 - 10x5, 0 - 8x5.	37µs
Endereça a CGRAM	0	0	0	1	A	A	A	A	A	A	Especifica um endereço (6 *bits*) de acesso à CGRAM.	37µs
Endereça a DDRAM	0	0	1	A	A	A	A	A	A	A	Especifica um endereço (7 *bits*) de acesso à DDRAM.	37µs
Lê contador de endereços e *Busy Flag*	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Lê o valor do contador de endereços (AC) e estado do *flag* BF: BF = 1, controlador ocupado BF = 0, controlador livre.	37µs
Escreve dado na CGRAM ou DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Escreve um dado (D) na posição indicada pelo registrador AC da memória CGRAM ou DDRAM.	37µs
Lê um dado da CGRAM ou DDRAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Lê um dado (D) da posição de memória (CGRAM ou DDRAM) endereçada pelo registrador AC.	37µs

Tabela 7-2

Repare que o envio de um comando deve obedecer aos seguintes passos:

1. O dado/comando é colocado no barramento de dados.
2. A linha RS é configurada para nível "1" caso seja um dado ou "0" caso seja um comando.
3. A linha E é pulsada.

A seqüência necessária para inicialização do módulo no modo de comunicação em 4 *bits* é:

1. É enviado um comando de quatro *bits* contendo o valor 3 (0011 binário) por meio das linhas DB4 a DB7.
2. Aguardam-se aproximadamente 5ms.
3. Envia-se novamente o comando do item 1.
4. Aguardam-se aproximadamente 100µs.
5. Envia-se novamente o comando do ítem 1.
6. Envia-se o comando 2 (0010 binário). Desse ponto em diante o módulo passa a trabalhar no modo de 4 *bits* e devemos enviar cada *byte* em duas metades de 4 *bits*, iniciando pelo *nibble* mais significativo.
7. Envia-se o comando de configuração de *display*.
8. Envia-se o comando de configuração liga/desliga do *display*.
9. Envia-se o comando de configuração de modo.

Observe que os dois últimos passos são opcionais e a sua ordem pode ser alterada.

Configurado o módulo, podemos escrever os dados nele a qualquer tempo. Basta enviar os dados com a linha RS em nível "1".

A referência bibiliográfica de número 40 apresenta um *link* para um *site* na Internet que contém um excelente tutorial sobre a programação de módulos LCD de caractere.

A utilização de um módulo LCD em conjunto com um microcontrolador MSP430 apresenta, a princípio, um obstáculo: tais módulos são normalmente alimentados com uma tensão de 5V, enquando os MSP430 podem operar, no máximo, a 3,6 Volts.

Essa incompatibilidade de níveis de tensão torna necessário adicionar um circuito conversor de níveis de tensão entre o microcontrolador e o módulo LCD. Existem diversos circuitos integrados disponíveis no mercado que permitem realizar esta tarefa, mas como normalmente, em um circuito microcontrolado, o custo e tamanho são fatores importantes, vamos demonstrar uma técnica de interface extremamente simples e barata.

Como já foi visto no tópico 5.4, os MSP430 possuem diodos internos de proteção (*clamping*) que protegem os circuitos das portas de E/S de níveis elevados de tensão externa (tanto acima do V_{DD} quanto abaixo do V_{SS}).

Isso significa que os pinos de E/S podem suportar tensões maiores que a de alimentação do *chip*, desde que a corrente não ultrapasse a admitida pelos diodos (cerca de 2mA). Além disso, a maioria dos dispositivos alimentados por 5V reconhecem como nível "1" uma tensão entre 2,2 e 5V. O MSP430 alimentado com 3,3V fornece uma tensão mínima de nível "1" igual a 3,05V, permitindo uma interface direta entre ambos, bastando a adição de um resistor em série com o pino, de forma a limitar a corrente que flui para dentro dele.

A figura 7-3 demonstra esta situação. Considerando que a tensão Vext seja de 5V, que a tensão V_{DD} seja igual a 3,3 Volts e que a tensão de junção dos diodos de proteção seja igual a 0,2 Volts, teremos uma tensão igual a 3,5 Volts na entrada do pino e a corrente circulante pelo resistor R será aproximadamente igual a I_R= 3,5 / R.

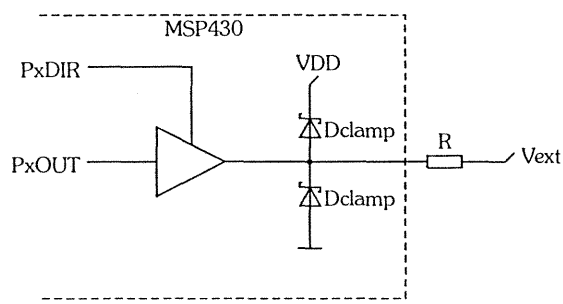


Figura 7-3

O valor da resistência R determina a corrente circulante. Neste caso, quanto maior o resistor menor a corrente, porém devemos lembrar que um resistor de valor muito elevado pode provocar uma queda de tensão muito grande quando o pino estiver operando como saída.

Neste caso, é necessário calcular o valor do resistor prevendo também a corrente de entrada do módulo LCD. Tipicamente, essa corrente é da ordem de µA.

Na placa Microlab X1, utilizam-se resistores de 2,2KΩ para essa função. O circuito de interface do módulo LCD com o microcontrolador pode ser visto na figura 7-4.

As listagens seguintes apresentam uma biblioteca ANSI C que pode ser utilizada para interface com módulos LCD. Nos exemplos, a linha LCD_CD (proveniente da linha R/S do módulo) foi conectada ao pino P2.1, a linha LCD_CE (proveniente da linha enable do módulo) foi conectada ao pino P2.0, a linha WRITE (proveniente do sinal R/W do módulo) foi conectada ao pino P2.2 do microcontrolador. As linhas de dados DB4 a DB7 foram conectadas aos pinos P1.4 a P1.7. As linhas DB0 a DB3 foram deixadas desconectadas.

O arquivo do exemplo 7-2 seguinte deve ser salvo como "lcdc.h", enquanto o arquivo do exemplo 7-3 deve ser salvo como "lcdc.c" na mesma pasta do projeto.

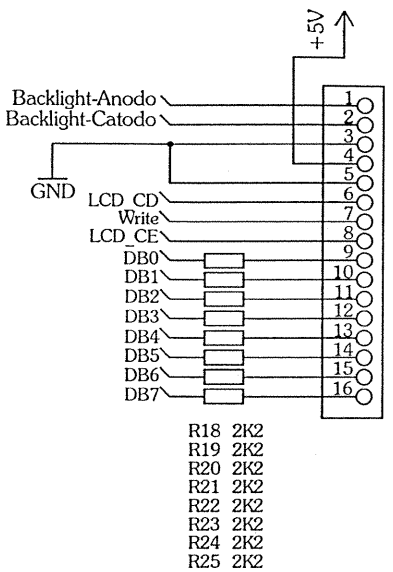


Figura 7-4

```
//*****************************************************************************
// Biblioteca de funções para manipulação de módulos LCD
//*****************************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este arquivo contém as definições e protótipos de funções de acesso e
// manipulação de módulos de display LCD que utilizam o controlador HD44780
// ou KS0066. As funções neste arquivo possuem o caráter didático e não foram
// otimizadas para aplicações reais.
//*****************************************************************************
// As definições a seguir são utilizadas para acesso aos pinos do  módulo
// Para redefinir estes pinos, basta declará-los no programa, antes da
// inclusão desta biblioteca
#ifndef lcdc_enable
   #define lcdc_enable      P2OUT_bit.P2OUT_0        // pino enable do LCD
   #define lcdc_enable_dir P2DIR_bit.P2DIR_0 // direção do pino enable
#endif
#ifndef lcdc_rs
   #define lcdc_rs  P2OUT_bit.P2OUT_1        // pino rs do LCD
   #define lcdc_rs_dir      P2DIR_bit.P2DIR_1        // direção do pino rs
#endif
#ifndef lcdc_d4
   #define lcdc_d4  P1OUT_bit.P1OUT_4        // pino de dados d4 do LCD
   #define lcdc_d4_dir      P1DIR_bit.P1DIR_4        // direção do pino d4
   #define lcdc_d5  P1OUT_bit.P1OUT_5        // pino de dados d5 do LCD
   #define lcdc_d5_dir      P1DIR_bit.P1DIR_5        // direção do pino d5
   #define lcdc_d6  P1OUT_bit.P1OUT_6        // pino de dados d6 do LCD
   #define lcdc_d6_dir      P1DIR_bit.P1DIR_6        // direção do pino d6
   #define lcdc_d7  P1OUT_bit.P1OUT_7        // pino de dados d7 do LCD
   #define lcdc_d7_dir      P1DIR_bit.P1DIR_7        // direção do pino d7
#endif
#define lcdc_seg_lin 0x40    // Endereço da segunda linha na RAM do LCD
// Definições utilizadas para configuração do display
#define lcdc_cursor_ligado 2
#define lcdc_cursor_desligado 0
#define lcdc_cursor_piscante 1
#define lcdc_cursor_fixo 0
#define lcdc_display_ligado 4
#define lcdc_display_desligado 0
#define lcdc_display_8x5 0
#define lcdc_display_10x5 4
#define lcdc_2_linhas 8
#define lcdc_1_linha 0
```

```
//*****************************************************************************
// Função de envio de um nibble para o display
//*****************************************************************************
// Argumentos de chamada:
// char dado : dado a ser enviado para o display (somente o nibble
// inferior)
//*****************************************************************************
// Retorno : nada
//*****************************************************************************
void lcdc_envia_nibble(char dado);

//*****************************************************************************
// Função de escrita de 1 byte no display
//*****************************************************************************
// Argumentos de chamada:
// char endereco : 0 se instrução, 1 se dado
// char dado : dado ou comando a ser escrito
//*****************************************************************************
// Retorno : nada
//*****************************************************************************
void lcdc_envia_byte(char endereco, char dado );

//*****************************************************************************
// Função de inicialização do display
//*****************************************************************************
// Argumentos de chamada:
// char modo1 : modo do display (número de linhas e tamanho dos caracteres
// char modo2 : modo do display (estado do cursor e do display)
//*****************************************************************************
// Retorno : nada
//*****************************************************************************
void lcdc_ini(char modo1, char modo2 );

//*****************************************************************************
// Função de posicionamento do cursor do display
//*****************************************************************************
// Argumentos de chamada:
// unsigned char x : coluna a ser posicionado o cursor (iniciando de 0)
// unsigned char y : linha a ser posicionado o cursor (0 ou 1)
//*****************************************************************************
// Retorno : nada
//*****************************************************************************
void lcdc_posiciona_texto(unsigned char x, unsigned char y);

//*****************************************************************************
// Função de escrita de um caractere no display
//*****************************************************************************
// Argumentos de chamada :
// unsigned char c : caractere a ser escrito
//*****************************************************************************
// Retorno : nada
//*****************************************************************************
// Observações :
// \f apaga o conteúdo do display
// \n e \r retornam o cursor para a primeira coluna da segunda linha
//*****************************************************************************
void lcdc_escreve_char(unsigned char c);

//*****************************************************************************
// Função de escrita de uma string no display
//*****************************************************************************
// Argumentos de chamada:
// unsigned char *c : um ponteiro para um caractere
//*****************************************************************************
// Retorno : nada
//*****************************************************************************
void lcdc_escreve_string (unsigned char *c);

//*****************************************************************************
// Função para ativar o display
//*****************************************************************************
// Argumentos de chamada : nenhum
// Retorno : nada
//*****************************************************************************
void lcdc_liga_display(void);
```

```
//*****************************************************************************
// Função para desativar o display
//*****************************************************************************
// Argumentos de chamada : nenhum
// Retorno : nada
//*****************************************************************************
void lcdc_desliga_display(void);

//*****************************************************************************
// Função para ativar o cursor
//*****************************************************************************
// Argumentos de chamada : nenhum
// Retorno : nada
//*****************************************************************************
void lcdc_liga_cursor(void);

//*****************************************************************************
// Função para desativar o cursor
//*****************************************************************************
// Argumentos de chamada : nenhum
// Retorno : nada
//*****************************************************************************
void lcdc_desliga_cursor(void);

//*****************************************************************************
// Função para ativar o cursor piscante
//*****************************************************************************
// Argumentos de chamada : nenhum
// Retorno : nada
//*****************************************************************************
void lcdc_liga_cursor_piscante(void);
//*****************************************************************************
// Função para desativar o cursor piscante
//*****************************************************************************
// Argumentos de chamada : nenhum
// Retorno : nada
//*****************************************************************************
void lcdc_desliga_cursor_piscante(void);
```

Exemplo 7-2

```
//*****************************************************************************
// Biblioteca de funções para manipulação de módulos LCD
//*****************************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este arquivo contém as funções de acesso e manipulação de módulos de dis-
// play LCD que utilizam o controlador HD44780 ou KS0066. As funções neste
// arquivo possuem o caráter didático e não foram otimizadas para aplicações
// reais.
//*****************************************************************************
// variável global que armazena o configuração do display
static char modo_lcd2;
union ubyte
{
   char _byte;
   struct
   {
      char b0 : 1;
      char b1 : 1;
      char b2 : 1;
      char b3 : 1;
      char b4 : 1;
      char b5 : 1;
      char b6 : 1;
      char b7 : 1;
   } bit;
};
void lcdc_delay_ms(unsigned int tempo)
{
  volatile unsigned int temp;
  for(;tempo;tempo--) for (temp=1000;temp;temp--);
}
```

```
void lcdc_envia_nibble(char dado)
{
   union ubyte teste;
   teste._byte = dado;
   // coloca os quatro bits nas saidas
   lcdc_d4 = teste.bit.b0;
   lcdc_d5 = teste.bit.b1;
   lcdc_d6 = teste.bit.b2;
   lcdc_d7 = teste.bit.b3;
   // pulsa a linha enable
   lcdc_enable = 1;
   lcdc_enable = 0;
}
void lcdc_envia_byte(char endereco, char dado )
{
   // coloca a linha rs em 0
   lcdc_rs = 0;
   // configura a linha rs dependendo do modo selecionado
   lcdc_rs = endereco;
   lcdc_delay_ms(1);
   // desativa linha enable
   lcdc_enable = 0;
   // envia a primeira parte do byte
   lcdc_envia_nibble(dado >> 4);
   // envia a segunda parte do byte
   lcdc_envia_nibble(dado & 0x0f);
}
void lcdc_ini(char modo1, char modo2 )
{
   char conta;
   // configura os pinos como saídas
   lcdc_enable_dir = 1;
   lcdc_rs_dir = 1;
   lcdc_d4_dir = 1;
   lcdc_d5_dir = 1;
   lcdc_d6_dir = 1;
   lcdc_d7_dir = 1;
   // coloca os pinos em nível lógico 0
   lcdc_d4 = 0;
   lcdc_d5 = 0;
   lcdc_d6 = 0;
   lcdc_d7 = 0;
   lcdc_rs = 0;
   lcdc_enable = 0;
   lcdc_delay_ms(15);
   // envia uma seqüência de 3 vezes 0x03
   // e depois 0x02 para configurar o módulo
   // para modo de 4 bits
   for(conta=1;conta<=3;++conta)
   {
      lcdc_envia_nibble(3);
      lcdc_delay_ms(5);
   }
   lcdc_envia_nibble(2);
   // envia códigos de inicialização do display
   lcdc_envia_byte(0,0x20 | modo1);
   lcdc_envia_byte(0,0x08 | modo2);
   modo_lcd2 = 0x08 | modo2;
   lcdc_envia_byte(0,1);
   lcdc_envia_byte(0,6);
}
void lcdc_posiciona_texto(unsigned char x, unsigned char y)
{
   char endereco;
   if (y) endereco = lcdc_seg_lin; else endereco = 0;
   endereco += x-1;
   lcdc_envia_byte(0,0x80|endereco);
}
void lcdc_escreve_char(unsigned char c)
{
   switch (c)
   {
      case '\f'          :   lcdc_envia_byte(0,1);
                             lcdc_delay_ms(5);
                             break;
```

```
        case '\n'          :
        case '\r'          :  lcdc_posiciona_texto(1,2);
                              break;
        default            :  lcdc_envia_byte(1,c);
    }
}
void lcdc_escreve_string (unsigned char *c)
{
    while (*c)
    {
        lcdc_escreve_char(*c);
        c++;
    }
}
void lcdc_liga_display(void)
{
    modo_lcd2 |= 4;
    lcdc_envia_byte (0,modo_lcd2);
}
void lcdc_desliga_display(void)
{
    modo_lcd2 &= 0xFB;
    lcdc_envia_byte (0,modo_lcd2);
}
void lcdc_liga_cursor(void)
{
    modo_lcd2 |= 2;
    lcdc_envia_byte (0,modo_lcd2);
}
void lcdc_desliga_cursor(void)
{
    modo_lcd2 &= 0xFD;
    lcdc_envia_byte (0,modo_lcd2);
}
void lcdc_liga_cursor_piscante(void)
{
    modo_lcd2 |= 1;
    lcdc_envia_byte (0,modo_lcd2);
}
void lcdc_desliga_cursor_piscante(void)
{
    modo_lcd2 &= 0xFE;
    lcdc_envia_byte (0,modo_lcd2);
}
```

Exemplo 7-3

```
#include <io430x14x.h>
#include "lcdc.h"
#include "lcdc.c"   // caso se utilize a biblioteca compilada esta linha
                    // pode ser removida
void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // desativa o watchdog
    // configura o DCO para a velocidade máxima
    DCOCTL = 255;
    BCSCTL1 = 7;
    // configura o pino P2.2 como saída e o coloca em nível "0"
    // o pino write do módulo deve ser conectado ao mesmo
    P2DIR_bit.P2DIR_2 = 1;
    P2OUT_bit.P2OUT_2 = 0;
    // inicializa o módulo
    lcdc_ini(lcdc_display_8x5|lcdc_2_linhas,lcdc_display_ligado|
            lcdc_cursor_desligado|lcdc_cursor_fixo);
    lcdc_escreve_char ('\f'); // apaga a tela
    // posiciona o cursor na sexta coluna da primeira linha
    lcdc_ posiciona_texto (5,0);
    lcdc_escreve_string("MSP430");    // escreve: MSP430
    // posiciona o cursor na primeira coluna da segunda linha
    lcdc_ posiciona_texto (0,1);
    lcdc_escreve_string("Teoria e Pratica");  // escreve: Teoria e Pratica
    while (1);
}
```

Exemplo 7-4

7.2.1. Voltímetro Digital Simples

O exemplo seguinte demonstra a leitura e apresentação da tensão lida do conversor ADC12.

Como utilizamos como referência a própria tensão de alimentação, o valor da tensão de entrada pode ser determinado da seguinte forma:

$$\text{tensão} = 3{,}3 * \frac{\text{ADC12MEM0}}{4095}$$

```
//*****************************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este programa implementa um voltímetro digital que apresenta a tensão lida
// em um módulo LCD 16x2. Foi utilizada a placa Microlab X1 com um módulo CPU
// MSP430F149. A tensão foi lida da entrada ANA2 conectada a um trimpot na
// própria placa.
//*****************************************************************************
#include <io430x14x.h>
#include "lcdc.h"
#include "lcdc.c"   // caso se utilize a biblioteca compilada esta linha
                    // pode ser removida
void main(void)
{
   unsigned long int temp;
   unsigned char string[10];
   WDTCTL = WDTPW + WDTHOLD; // desativa o watchdog
   // configura o DCO para a velocidade máxima
   DCOCTL = 255;
   BCSCTL1 = 7;
   // configura o pino P2.2 como saída e o coloca em nível "0"
   // o pino write do módulo deve ser conectado ao mesmo
   P2DIR_bit.P2DIR_2 = 1;
   P2OUT_bit.P2OUT_2 = 0;
   // configura o ADC12 para medição contínua da memória 0 no
   // primeiro canal de entrada
   ADC12CTL1 = CONSEQ_2 + SHP;
   ADC12MCTL0 = 0;
   ADC12CTL0 = MSC + ADC12ON + ENC + ADC12SC;
   P6DIR_bit.P6DIR_0 = 0;    // pino P6.0 como entrada
   P6SEL_bit.P6SEL_0 = 1;    // pino P6.0 na função ADC
   // inicializa o módulo  lcdc_ini(lcdc_display_8x5|lcdc_2_linhas,lcdc_display_ligado|
           lcdc_cursor_desligado|lcdc_cursor_fixo);
   lcdc_escreve_char ('\f'); // apaga a tela
   // posiciona o cursor na segunda coluna da primeira linha
   lcdc_posiciona_texto(1,0);
   lcdc_escreve_string("Tensao=");
   while (1)
   {
      // Lê e ajusta a tensão
      temp = (long)ADC12MEM0 * 3300 / 4095;
      // converte para uma string
      sprintf(string,"%04ld",temp);
      string[5]=0;
      string[4]=string[3];
      string[3]=string[2];
      string[2]=string[1];
      string[1]='.';
      lcdc_posiciona_texto(8,1);
      lcdc_escreve_string(string);   // escreve o valor da tensão
      lcdc_escreve_string("V");
   }
}
```

Exemplo 7-5

7.2.2. Termômetro Digital

No exemplo seguinte, demonstramos a utilização do sensor de temperatura integrado em alguns modelos dos MSP430.

A temperatura pode ser determinada pela seguinte fórmula:

$$\text{TEMP}(^{\circ}\text{C}) = \frac{V_{\text{TEMP}} - 0{,}986}{0{,}00355}$$

Utilizando a referência interna de 1,5V, podemos determinar a temperatura em função do resultado binário do ADC12 pela seguinte fórmula:

$$\text{TEMP}(^{\circ}\text{C}) = \frac{\frac{1{,}5 * \text{ADC12MEM0}}{4095} - 0{,}986}{0{,}00355} = \left(\frac{1{,}5 * \text{ADC12MEM0}}{4095} - 0{,}986\right) * 282$$

$$\text{TEMP}(^{\circ}\text{C}) = \frac{\text{ADC12MEM0} * 423}{4095} - 278$$

Para simplificar os cálculos, podemos utilizar o valor 4.096 em vez de 4.095, pois a divisão por 4.096 pode ser realizada por uma operação de rotação de 12 *bits* à direita. Para aumentar a resolução do valor apresentado, multiplicamos as contantes 423 e 278 por dez e obtemos a seguinte fórmula final:

$$\text{TEMP}(^{\circ}\text{C}) \cong \frac{\text{ADC12MEM0} * 4230}{4096} - 2780$$

```
//*****************************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este programa implementa um termometro digital que apresenta a temperatura
// lida em um módulo LCD 16x2. Foi utilizada a placa Microlab X1 com um módu-
// lo CPU MSP430F149. A temperatura é lida através do sensor interno do ADC12
//*****************************************************************************
#include <io430x14x.h>
#include <stdio.h>
#include "lcdc.h"
#include "lcdc.c"   // caso se utilize a biblioteca compilada esta linha
                    // pode ser removida
void main(void)
{
   unsigned long int temperatura;
   unsigned int buffer[8], offset;
   unsigned char string[10], temp;
   WDTCTL = WDTPW + WDTHOLD; // desativa o watchdog
   // configura o DCO para a velocidade máxima
   DCOCTL = 255;
   BCSCTL1 = 7;
   // configura o pino P2.2 como saída e o coloca em nível "0"
   // o pino write do módulo deve ser conectado ao mesmo
   P2DIR_bit.P2DIR_2 = 1;
   P2OUT_bit.P2OUT_2 = 0;
   // configura o ADC12 para medição contínua da memória 0
   ADC12CTL1 = CONSEQ_2 + SHP + ADC12DIV_7;
   // seleciona o diodo de temperatura e a referência Vref+
   // para a memória 0
   ADC12MCTL0 = SREF_1 + INCH_10;
```

```
    // seleciona um período de amostragem de 1024 ciclos de clock
    // do ADC12, modo contínuo
    ADC12CTL0 = SHT0_15 + MSC + REFON + ADC12ON + ENC + ADC12SC;
    // inicializa o módulo  lcdc_ini(lcdc_display_8x5|lcdc_2_linhas,lcdc_display_ligado|
            lcdc_cursor_desligado|lcdc_cursor_fixo);
    lcdc_escreve_char ('\f'); // apaga a tela
    // posiciona o cursor na primeira coluna da primeira linha
    lcdc_posiciona_texto(1,0);
    lcdc_escreve_string("Temp=");
    offset = 100;
    for (temp=0; temp<8; temp++) buffer[temp]=0;
    while (1)
    {
        // lê o resultado da conversão e calcula a temperatura
        temperatura = (((long)ADC12MEM0*4230)>>12)-2780;
        // faz a média das últimas 8 leituras
        for (temp=0; temp<7; temp++) buffer[temp]=buffer[temp+1];
        buffer[7]= temperatura+offset;
        for (temp=0; temp<7; temp++) temperatura += buffer[temp];
        temperatura >>= 3;     // divide por 8
        // converte o resultado em uma string de 5 dígitos
        sprintf(string,"%5ld",temperatura);
        // desloca os 3 dígitos mais significativos para colocar o ponto decimal
        string[0]=string[1];
        string[1]=string[2];
        string[2]=string[3];
        string[3]='.';
        lcdc_posiciona_texto(6,0);
        lcdc_escreve_string(string); // escreve o resultado no display
        lcdc_escreve_char(0xDF);       // símbolo de grau
        lcdc_escreve_char('C');
        // um atraso utilizando a própria variável temperatura
        for (temperatura=300000;temperatura;temperatura--);
    }
}
```

Exemplo 7-6

7.3. Display LCD Gráfico

Neste tópico vamos demonstrar a programação e utilização de um módulo LCD gráfico baseado no controlador T6963C da Toshiba.

Esse controlador, projetado para ser controlado diretamente por um microprocessador compatível com Z-80, possui um barramento de 8 *bits* bidirecional, pelo qual é possível trocar dados e comandos, além de uma linha de seleção de escrita ($\overline{WR}$), outra de seleção de leitura ($\overline{RD}$), habilitação do *chip* ($\overline{CE}$) e seleção entre dados ou comandos (C/$\overline{D}$). Além desses pinos, outros também estão disponíveis, mas não são necessários na comunicação entre o microcontrolador e o módulo.

O módulo utilizado no exemplo (G241281BNHDWB fabricado pela Palmtech) possui uma resolução de 240x128 *pixels*, com caracteres brancos sobre fundo azul, *backlight* a LED (azul) e não possui gerador interno de tensão de contraste, devendo esta ser fornecida externamente pelo pino Vo. Ele possui ainda uma memória RAM de 8*Kbytes* utilizada para armazenar tanto textos quanto gráficos.

A comunicação com o controlador T6963C não apresenta grandes problemas, a não ser o fato de ser alimentado com uma tensão de 5V, razão pela qual utilizamos a mesma solução de interface utilizada com o módulo LCD de caractere.

Além disso, utilizamos a checagem de estado do controlador, de forma a nos certificarmos de somente enviar comandos ou dados quando ele estiver pronto para recebê-los e processá-los. O registrador de estado do controlador pode ser lido colocando a linha C/$\overline{D}$ em nível "1",

$\overline{CE}$ em "0" e efetuando uma operação de leitura das linhas de dados. Cada um dos bits possui um significado, a saber:

Bit 7	*Bit 6*	*Bit 5*	*Bit 4*	*Bit 3*	*Bit 2*	*Bit 1*	*Bit 0*
STA7	STA6	STA5	STA4	STA3	STA2	STA1	STA0

- STA0 - capacidade de execução de comandos;
- STA1 - capacidade de leitura/escrita;
- STA2 - capacidade de leitura no modo auto;
- STA3 - capacidade de escrita no modo auto;
- STA5 - capacidade de operação do controlador;
- STA6 - indicador de erro nas operações de leitura e escrita da tela;
- STA7 - indicador da condição de piscagem.

Os comandos são enviados colocando a linha $C/\overline{D}$ em nível "1", $\overline{CE}$ em "0" e efetuando uma operação de escrita. No caso de existirem parâmetros, primeiramente é transferido o *byte* D1, em seguida D2 (caso exista) e depois o comando. Antes do envio de cada *byte* é necessário verificar se o controlador está pronto para receber o comando (*bits* de estado STA1 e STA0 em nível "1"). O controlador pode interpretar um dos seguintes comandos:

Comando	Comando								D1	D2	Função
	Bit 7	*Bit 6*	*Bit 5*	*Bit 4*	*Bit 3*	*Bit 2*	*Bit 1*	*Bit 0*			
Setar apontador do cursor	0	0	1	0	0	0	0	1	endereço x	endereço y	-
Setar registrador de offset	0	0	1	0	0	0	1	0	dado	0x00	-
Setar o apontador de endereço	0	0	1	0	0	1	0	0	LSB endereço	MSB endereço	-
Setar o endereço inicial da área de texto	0	1	0	0	0	0	0	0	LSB endereço	MSB endereço	-
Setar área de texto	0	1	0	0	0	0	0	1	no. de colunas	0x00	Configura o número de colunas da tela de texto
Setar o endereço inicial da área de gráficos	0	1	0	0	0	0	1	0	LSB endereço	MSB endereço	
Setar área de gráficos	0	1	0	0	0	0	1	1	no. de colunas	0x00	Configura o número de colunas da tela gráfica
Seleção do modo CG-ROM interna	1	0	0	0	0	0	0	0	-	-	Modo OU
	1	0	0	0	0	0	0	1	-	-	Modo EOU
	1	0	0	0	0	0	1	1	-	-	Modo E
	1	0	0	0	0	1	0	0	-	-	Modo de atributos de texto
Seleção do modo CG-RAM externa	1	0	0	0	1	0	0	0	-	-	Modo OU
	1	0	0	0	1	0	0	1	-	-	Modo EOU
	1	0	0	0	1	0	1	1	-	-	Modo E
	1	0	0	0	1	1	0	0	-	-	Modo de atributos de texto
Modo do *display*	1	0	0	1	0	0	0	0	-	-	*Display* desligado
	1	0	0	1	x	x	1	0	-	-	Cursor ligado, não piscante
	1	0	0	1	x	x	1	1	-	-	Cursor ligado, não piscante
	1	0	0	1	0	1	x	x	-	-	Texto ligado, gráfico desligado
	1	0	0	1	1	0	x	x	-	-	Texto desligado, gráfico ligado
	1	0	0	1	1	1	x	x	-	-	Texto ligado, gráfico ligado

Comando	Comando								D1	D2	Função
	Bit 7	*Bit* 6	*Bit* 5	*Bit* 4	*Bit* 3	*Bit* 2	*Bit* 1	*Bit* 0			
Seleção do padrão do cursor	1	0	1	0	0	0	0	0	-	-	cursor de 1 linha
	1	0	1	0	0	0	0	1	-	-	cursor de 2 linhas
	1	0	1	0	0	0	1	0	-	-	cursor de 3 linhas
	1	0	1	0	0	0	1	1	-	-	cursor de 4 linhas
	1	0	1	0	0	1	0	0	-	-	cursor de 5 linhas
	1	0	1	0	0	1	0	1	-	-	cursor de 6 linhas
	1	0	1	0	0	1	1	0	-	-	cursor de 7 linhas
	1	0	1	0	0	1	1	1	-	-	cursor de 8 linhas
Leitura/escrita de dados no modo auto	1	0	1	1	0	0	0	0	-	-	Inicia modo auto para escrita
	1	0	1	1	0	0	0	1	-	-	Inicia modo auto para leitura
	1	0	1	1	0	0	1	0	-	-	Sai do modo auto
Leitura/escrita de dados	1	1	0	0	0	0	0	0	dado	-	Escreve dado e incrementa o apontador de endereço
	1	1	0	0	0	0	0	1	-	-	Lê dado e incrementa o apontador de endereço
	1	1	0	0	0	0	1	0	dado	-	Escreve dado e decrementa o apontador de endereço
	1	1	0	0	0	0	1	1	-	-	Lê dado e decrementa o apontador de endereço
	1	1	0	0	0	1	0	0	dado	-	Escreve dado
	1	1	0	0	0	1	0	1	-	-	Lê dado
Lê tela	1	1	1	0	0	0	0	0	-	-	
Copia tela	1	1	1	0	1	0	0	0			
Seta *bit*	1	1	1	1	1	B	B	B	-	-	Seta o *bit* especificado por B
Apaga *bit*	1	1	1	1	0	B	B	B	-	-	Apaga o *bit* especificado por B

Tabela 7-3

A figura 7-5 ilustra o conjunto de caracteres disponível na ROM do gerador de caracteres que integra o controlador.

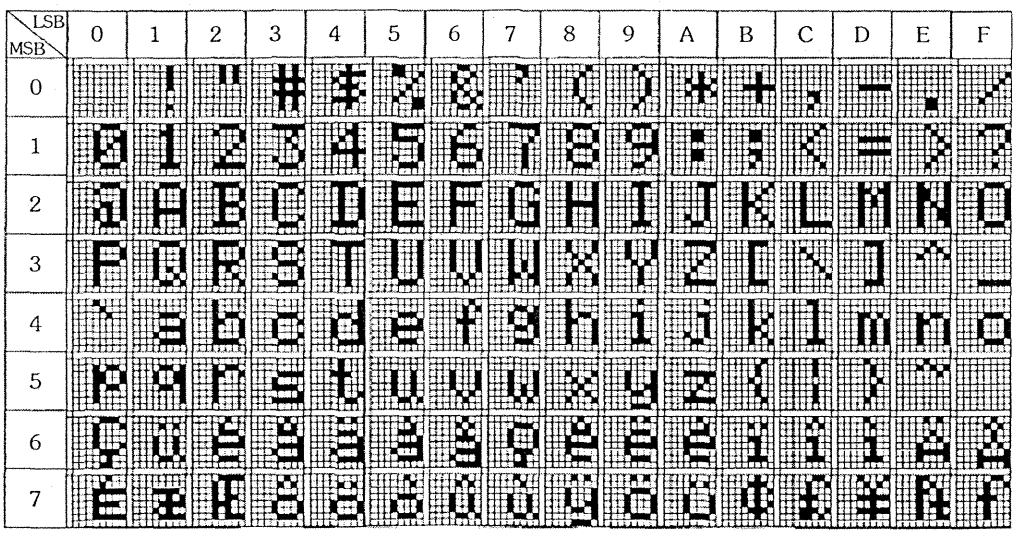


Figura 7-5

A configuração do controlador é relativamente complexa e não vamos nos estender mais neste assunto. A biblioteca apresentada a seguir é auto-explicativa e deve servir como ponto de partida para o desenvolvimento de aplicações que utilizem esse tipo de módulo de *display*.

```
//*****************************************************************************
// Biblioteca de funções para manipulação de módulos LCD gráficos
//*****************************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este arquivo contém as definições e protótipos de funções de acesso e ma-
// nipulação de módulos de display LCD gráfico que utilizam o controlador
// T6963C. As funções neste arquivo possuem o caráter didático e não foram
// otimizadas para aplicações reais.
//*****************************************************************************

// As definições a seguir são utilizadas para acesso aos pinos do
// módulo LCD gráfico
// Para redefinir estes pinos, basta declará-los no programa, antes
// da inclusão desta biblioteca
#ifndef lcdg_cd
   #define lcdg_cd P2OUT_bit.P2OUT_1
   #define lcdg_cd_dir P2DIR_bit.P2DIR_1
#endif
#ifndef lcdg_rd
   #define lcdg_rd P2OUT_bit.P2OUT_3
   #define lcdg_rd_dir P2DIR_bit.P2DIR_3
#endif
#ifndef lcdg_wr
   #define lcdg_wr P2OUT_bit.P2OUT_2
   #define lcdg_wr_dir P2DIR_bit.P2DIR_2
#endif
#ifndef lcdg_ce
   #define lcdg_ce P2OUT_bit.P2OUT_0
   #define lcdg_ce_dir P2DIR_bit.P2DIR_0
#endif
#ifndef lcdg_dados_out
   #define lcdg_dados_dir P1DIR
   #define lcdg_dados_out P1OUT
   #define lcdg_dados_in P1IN
#endif
// definições do display gráfico
#define lcdg_largura_caractere 6              // 6 = 6x8  - 8 = 8x8
#define lcdg_pixels_x 240                     // largura do display em pixels
#define lcdg_pixels_y 128                     // altura do display em pixels
#define lcdg_memoria_ram 8192                 // capacidade de memória RAM
#define lcdg_inicio_area_grafica 0
#define lcdg_bytes_por_linha_grafica lcdg_pixels_x/lcdg_largura_caractere
#define lcdg_bytes_por_linha_texto lcdg_bytes_por_linha_grafica
#define lcdg_inicio_area_texto lcdg_memoria_ram-
((lcdg_pixels_y/8)*lcdg_bytes_por_linha_texto)
#define lcdg_auto_write_mode 0xB0
#define lcdg_auto_read_mode 0xB1
#define lcdg_desliga_auto_mode 0xB2
#define lcdg_modo_OR 0x80
#define lcdg_modo_XOR 0x81
#define lcdg_modo_AND 0x82

//*****************************************************************************
// lcdg_aguarda_status
//*****************************************************************************
// Entrada: unsigned char valor - valor de estado a ser comparado
// Retorna: void
//*****************************************************************************
// Aguarda que os bits de status especificados sejam apagados.
//*****************************************************************************
void lcdg_aguarda_status(unsigned char valor);

//*****************************************************************************
// lcdg_escreve_comando
//*****************************************************************************
// Entrada: unsigned char comando - comando a ser enviado
// Retorna: void
//*****************************************************************************
```

```
// Envia um comando para o display
//***************************************************************************
void lcdg_escreve_comando(unsigned char comando);

//***************************************************************************
// lcdg_escreve_byte
//***************************************************************************
// Entrada: unsigned char dado - informação a ser enviada para o LCD
// Retorna: void
//***************************************************************************
// Envia um dado para o display
//***************************************************************************
void lcdg_escreve_byte(unsigned char dado);

//***************************************************************************
// lcdg_escreve_byte_auto
//***************************************************************************
// Entrada: unsigned char dado - informação a ser enviada para o LCD
// Retorna: void
//***************************************************************************
// Envia um dado para o display no modo automático (transferência de um bloco
// de dados)
//***************************************************************************
void lcdg_escreve_byte_auto(unsigned char dado);

//***************************************************************************
// lcdg_escreve_word
//***************************************************************************
// Entrada: unsigned int dado - informação a ser enviada para o LCD
// Retorna: void
//***************************************************************************
// Envia um dado de 16 bits para o display
//***************************************************************************
void lcdg_escreve_word(unsigned int dado);

//***************************************************************************
// lcdg_seta_address_pointer
//***************************************************************************
// Entrada: unsigned int end - configura o apontador de endereços do
//                                  display
// Retorna: void
//***************************************************************************
// Envia um dado de 16 bits para o display
//***************************************************************************
void lcdg_seta_address_pointer(unsigned int end);

//***************************************************************************
// lcdg_apaga_tela_grafica
//***************************************************************************
// Entrada: void
// Retorna: void
//***************************************************************************
// Apaga a informação da área gráfica do display, preenchendo toda a área
// gráfico com o valor 0, utilizando o modo automático
//***************************************************************************
void lcdg_apaga_tela_grafica(void);

//***************************************************************************
// lcdg_apaga_tela_texto
//***************************************************************************
// Entrada: void
// Retorna: void
//***************************************************************************
// Apaga a informação da área de texto do display, preenchendo toda a área de
// texto com zero, utilizando o modo automático
//***************************************************************************
void lcdg_apaga_tela_texto(void);

//***************************************************************************
// lcdg_ini
//***************************************************************************
// Entrada: void
// Retorna: void

//***************************************************************************
// Inicializa o display
```

```
//*****************************************************************************
void lcdg_ini(void);

//*****************************************************************************
// lcdg_seta_modo
//*****************************************************************************
// Entrada: unsigned char modo - modo do display
// Retorna: void
//*****************************************************************************
// Seleciona um dos modos de interação entre a área gráfica e a área de
// texto:
// OR       - operação OR entre os pontos gráficos e de texto
// XOR      - operação XOR entre os pontos gráficos e de texto
// AND      - operação AND entre os pontos gráficos e de texto
//*****************************************************************************
void lcdg_seta_modo(unsigned char modo);

//*****************************************************************************
// lcdg_posiciona_texto
//*****************************************************************************
// Entrada: unsigned char col - coluna da tela de texto
//          unsigned char linha - linha da tela de texto
// Retorna: void
//*****************************************************************************
// Posiciona o cursor para a escrita de caracteres no display
//*****************************************************************************
void lcdg_posiciona_texto(unsigned char col,unsigned char linha);

//*****************************************************************************
// lcdg_escreve_char
//*****************************************************************************
// Entrada: unsigned char dado - caractere a ser escrito
// Retorna: void
//*****************************************************************************
// Escreve um caractere na área indicada pelo cursor
//*****************************************************************************
void lcdg_escreve_char(unsigned char dado);

//*****************************************************************************
// lcdg_escreve_string
//*****************************************************************************
// Entrada: unsigned char dado[] - ponteiro para uma string de caracteres
//                                 terminada por nulo
// Retorna: void
//*****************************************************************************
// Escreve uma string de caracteres no local apontado pelo cursor
//*****************************************************************************
void lcdg_escreve_string(unsigned char dado[]);

//*****************************************************************************
// lcdg_pixel
//*****************************************************************************
// Entrada: unsigned char x - coluna da tela gráfica
//          unsigned char y - linha da tela gráfica
//          unsigned char cor - 0 - apagado, 1 - aceso
// Retorna: void
//*****************************************************************************
// Liga ou desliga um pixel na tela gráfica
//*****************************************************************************
void lcdg_pixel(unsigned char x, unsigned char y, unsigned char cor);

//*****************************************************************************
// lcdg_linha
//*****************************************************************************
// Entrada: unsigned char x1 - coluna inicial
//          unsigned char y1 - linha inicial
//          unsigned char x2 - coluna final
//          unsigned char y2 - linha final
//          unsigned char cor - 0 - apagado, 1 - aceso
// Retorna: void
//*****************************************************************************
// Liga ou desliga um pixel na tela gráfica
//*****************************************************************************
void lcdg_linha(unsigned char x1, unsigned char y1, unsigned char x2,
                unsigned char y2, unsigned char color);
```

```
//*****************************************************************************
// lcdg_retangulo
//*****************************************************************************
// Entrada: unsigned char x1 - coluna inicial
//           unsigned char y1 - linha inicial
//           unsigned char x2 - coluna final
//           unsigned char y2 - linha final
//           unsigned char cor - 0 - apagado, 1 - aceso
// Retorna: void
//*****************************************************************************
// Desenha um quadrado/retângulo na tela
//*****************************************************************************
void lcdg_retangulo(unsigned char x0, unsigned char y0,
                unsigned char x1, unsigned char y1, unsigned char cor);

//*****************************************************************************
// lcdg_retangulo_preenchido
//*****************************************************************************
// Entrada: unsigned char x1 - coluna inicial
//           unsigned char y1 - linha inicial
//           unsigned char x2 - coluna final
//           unsigned char y2 - linha final
//           unsigned char cor - 0 - apagado, 1 - aceso
// Retorna: void
//*****************************************************************************
// Desenha um quadrado/retângulo totalmente preenchido na tela
//*****************************************************************************
void lcdg_retangulo_preenchido(unsigned char x0, unsigned char y0,
           unsigned char x1, unsigned char y1, unsigned char cor);

//*****************************************************************************
// lcdg_escreve_tela
//*****************************************************************************
// Entrada: unsigned char *matriz - ponteiro para a matriz com os
//           elementos da tela
// Retorna: void
//*****************************************************************************
// Copia uma matriz da memória para a tela do display
//*****************************************************************************
void lcdg_escreve_tela (const unsigned char *matriz);
```

Exemplo 7-7

```
//*****************************************************************************
// Biblioteca de funções para manipulação de módulos LCD gráficos
//*****************************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este arquivo contém as funções de acesso e manipulação de módulos de dis-
// play LCD gráfico que utilizam o controlador T6963C. As funções neste ar-
// quivo possuem o caráter didático e não foram otimizadas para aplicações
// reais.
//*****************************************************************************
void lcdg_aguarda_status(unsigned char valor)
{
   unsigned char result = 0;
   while (!result)
   {
      lcdg_dados_dir = 0;    // configura os dados como entradas
      lcdg_cd = 1;           // linha CD = 1 (modo comando)
      lcdg_rd = 0;           // seleciona leitura
      lcdg_wr = 1;
      lcdg_ce = 0;           // habilita o controlador
      // lê o status e compara com o valor, se igual, result = 1
      // se diferente, result = 0
      if ((lcdg_dados_in & valor)==valor) result = 1; else result = 0;
      lcdg_ce = 1;           // desabilita o controlador
   }
}
void lcdg_escreve_comando(unsigned char comando)
{
   // aguarda o controlador estar livre (STA1 = STA0 = 1)
   lcdg_aguarda_status(3);
```

```
    // coloca o comando na porta de saída
    lcdg_dados_out = comando;
    lcdg_cd = 1;      // modo comando
    lcdg_wr = 0;      // seleciona escrita
    lcdg_rd = 1;
    // configura as linhas de dados como saídas
    lcdg_dados_dir = 0xFF;
    lcdg_ce = 0;      // habilita o controlador
    // o T6963C efetua a leitura do comando nesse momento
    lcdg_ce = 1;      // desabilita o controlador
}
void lcdg_escreve_byte(unsigned char dado)
{
    // aguarda o controlador estar livre (STA1 = STA0 = 1)
    lcdg_aguarda_status(3);
    // carrega a porta de saída com o dado a ser escrito
    lcdg_dados_out = dado;
    lcdg_cd = 0;      // modo de dados
    lcdg_wr = 0;      // seleciona escrita
    lcdg_rd = 1;
    // configura a porta como saída
    lcdg_dados_dir = 0xFF;
    lcdg_ce = 0;      // habilita o controlador
    // o T6963C efetua a leitura do dado nesse momento
    lcdg_ce = 1;      // desabilita o controlador
}
void lcdg_escreve_byte_auto(unsigned char dado)
{
    // aguarda o controlador estar livre (STA3 = 1)
    lcdg_aguarda_status(8);
    // coloca o dado na porta de saída
    lcdg_dados_out = dado;
    lcdg_cd = 0;      // modo de dados
    lcdg_wr = 0;      // seleciona escrita
    lcdg_rd = 1;
    // configura a porta como saída
    lcdg_dados_dir = 0xFF;
    lcdg_ce = 0;      // habilita o controlador
    // o T6963C efetua a leitura do dado nesse momento
    lcdg_ce = 1;      // desabilita o controlador
}
void lcdg_escreve_word(unsigned int dado)
{
    // aguarda o controlador estar livre (STA1 = STA0 = 1)
    lcdg_aguarda_status(3);
    // escreve a parte baixa do dado
    lcdg_escreve_byte(dado);
    // escreve a parte alta do dado
    lcdg_escreve_byte(dado>>8);
}
void lcdg_seta_address_pointer(unsigned int end)
{
    // aguarda o controlador estar livre (STA1 = STA0 = 1)
    lcdg_aguarda_status(3);
    // envia o endereço para o controlador
    lcdg_escreve_word(end);
    // envia o comando
    lcdg_escreve_comando(0x24);
}
void lcdg_apaga_tela_grafica(void)
{
    unsigned int temp;
    // configura o apontador de endereço para o início da área gráfica
    lcdg_seta_address_pointer(lcdg_inicio_area_grafica);
    // seleciona o modo de escrita automática
    lcdg_escreve_comando(lcdg_auto_write_mode);
    // o laço for preenche a área gráfica
    for (temp = (lcdg_bytes_por_linha_grafica * lcdg_pixels_y);temp;temp--)
        lcdg_escreve_byte_auto(0);     // escreve o valor zero
    // desliga o modo auto
    lcdg_escreve_comando(lcdg_desliga_auto_mode);
}
void lcdg_apaga_tela_texto(void)
{
    unsigned int temp;
    // configura o apontador de endereço para o início da área de texto
```

```
    lcdg_seta_address_pointer(lcdg_inicio_area_texto);
    // seleciona o modo de escrita automática
    lcdg_escreve_comando(lcdg_auto_write_mode);
    // o laço for preenche a área gráfica
    for (temp =(lcdg_bytes_por_linha_texto*(lcdg_pixels_y>>3));temp;temp--)
        lcdg_escreve_byte_auto(0);     // escreve o valor zero
    // desliga o modo auto
    lcdg_escreve_comando(lcdg_desliga_auto_mode);
}

void lcdg_ini(void)
{
    // configura os pinos de controle como saídas
    lcdg_wr_dir = 1;
    lcdg_rd_dir = 1;
    lcdg_ce_dir = 1;
    lcdg_cd_dir = 1;
    // seleciona a CG-ROM
    lcdg_escreve_comando(0x80);
    // configura o início da área gráfica
    lcdg_escreve_word(lcdg_inicio_area_grafica);
    lcdg_escreve_comando(0x42);
    // configura o número de bytes de uma linha gráfica completa
    lcdg_escreve_byte(lcdg_bytes_por_linha_grafica);
    lcdg_escreve_byte(0);
    lcdg_escreve_comando(0x43);
    // configura o início da área de texto
    lcdg_escreve_word(lcdg_inicio_area_texto);
    lcdg_escreve_comando(0x40);
    // configura o número de bytes de uma linha de texto
    lcdg_escreve_byte(lcdg_bytes_por_linha_texto);
    lcdg_escreve_byte(0);
    lcdg_escreve_comando(0x41);
    // seleciona o modo OU
    lcdg_escreve_comando(0x80);
    // seleciona o modo com texto e gráfico ligados
    lcdg_escreve_comando(0x9C);
    lcdg_apaga_tela_grafica();        // apaga a tela gráfica
    lcdg_apaga_tela_texto();          // apaga a tela de texto
}

void lcdg_seta_modo(unsigned char modo)
{
  lcdg_escreve_comando(modo);
}

void lcdg_posiciona_texto(unsigned char col,unsigned char linha)
{
    lcdg_seta_address_pointer(lcdg_inicio_area_texto + ((int)linha *
lcdg_bytes_por_linha_texto) + col);
}

void lcdg_escreve_char(unsigned char dado)
{
    // o conjunto de caracteres do display está deslocado 32 posições
    // em relação à tabela ASCII, por isso, subtraímos este valor do
    // código do caractere a ser escrito
    lcdg_escreve_byte(dado-32);
    // escreve o dado e incrementa o apontador de endereço
    lcdg_escreve_comando(0xC0);
}

void lcdg_escreve_string(unsigned char dado[])
{
    unsigned char temp=0;
    // enquanto não encontra um caractere nulo
    while (dado[temp])
    {
        // escreve o caractere da posição atual da matriz
        lcdg_escreve_char(dado[temp]);
        // incrementa o apontador
        temp++;
    }
}
void lcdg_pixel(unsigned char x, unsigned char y, unsigned char cor)
{
```

```
    // a função de plotar um pixel na tela funciona de forma diferente
    // nos modos 6x8 ou 8x8
    // no modo 8x8 cada byte da memória guarda 8 pixels
    #if lcdg_largura_caractere==8
        unsigned char temp,end,temp2;
        // primeiro divide o valor x por 8
        end = x>>1 ;  // divide por dois
        end >>= 2;              // divide por quatro
        // o resultado da divisão é multiplicado por 8
        temp2 = end<<1;         // multiplica por dois
        temp2 <<= 2;            // multiplica por quatro
        // dependendo do pixel estar aceso ou apagado
        // é calculado o valor do comando a ser enviado ao display
        // a ordem dos pixels na memória é inversa ao eixo x:
        // pixel 0,0 = bit 7 do endereço 0
        // pixel 1,0 = bit 6 do endereço 0 e assim por diante
        if (cor) temp = 0xF7-(x-temp2); else temp = 0xFF-(x-temp2);
    #else
        // no modo 6x8, cada byte da memória guarda 6 pixels
        // os bits 7 e 6 são mantidos em zero
        unsigned char temp,end;
        end = x / 6;  // divide o valor de x por 6
        // dependendo do pixel estar aceso ou apagado
        // é calculado o valor do comando a ser enviado ao display
        // a ordem dos pixels na memória é inversa ao eixo x:
        // pixel 0,0 = bit 5 do endereço 0
        // pixel 1,0 = bit 4 do endereço 0 e assim por diante
        if (cor) temp = 0xFD-(x-end*6); else temp = 0xF5-(x-end*6);
    #endif
    // configura o apontador de endereço do display
    lcdg_seta_address_pointer(lcdg_inicio_area_grafica+end+
            lcdg_bytes_por_linha_grafica*y);
    // escreve o comando de ligar/desligar pixel
    lcdg_escreve_comando(temp);
}
void lcdg_linha(unsigned char x1, unsigned char y1, unsigned char x2,
                    unsigned char y2, unsigned char color)
{
    // Esta função utiliza o algoritmo de Bresenham para o cálculo
    // da reta
    signed int  addx, addy, dx, dy, erro;
    unsigned char x,y,i;
    // calcula o módulo das distâncias Dx e Dy
    if (x2>=x1) dx=x2-x1; else dx=x1-x2;
    if (y2>=y1) dy=y2-y1; else dy=y1-y2;
    // ponto inicial da linha
    x = x1;
    y = y1;
    if(x1 > x2) addx = -1; else addx = 1;
    if(y1 > y2) addy = -1; else addy = 1;
    if (!dx) // linha vertical
        if (addy>0) for( i=y1; i <= y2; i++ ) lcdg_pixel(x, i, color);
            else for( i=y2; i <= y1; i++ ) lcdg_pixel(x, i, color);
    else if (!dy)    // linha horizontal
        if (addx>0)   for( i=x1; i <= x2; i++ ) lcdg_pixel(i, y, color);
            else for( i=x2; i <= x1; i++ ) lcdg_pixel(i, y, color);
    else
    // linha inclinada
    if(dx >= dy)
    {
        // calcula o fator de erro
        erro = (dy<<1) - dx;
        for(i=0; i<=dx; ++i)
        {
            lcdg_pixel(x, y, color);    // plota o ponto
            if(erro < 0)
            {
                erro += dy<<1;
                x += addx;
            } else
            {
                erro += (dy - dx)<<1;
                x += addx;
                y += addy;
            }
        }
```

```
    } else
    {
        // calcula o fator de erro
        erro = (dx<<1) - dy;
        for(i=0; i<=dy; ++i)
        {
            lcdg_pixel(x, y, color);    // plota o ponto
            if(erro < 0)
            {
                erro += dx<<1;
                y += addy;
            } else
            {
                erro += (dx - dy)<<1;
                x += addx;
                y += addy;
            }
        }
    }
}
void lcdg_retangulo(unsigned char x0, unsigned char y0, unsigned char x1, unsigned char
y1, unsigned char cor)
{
  lcdg_linha(x0,y0,x1,y0,cor);
  lcdg_linha(x1,y0,x1,y1,cor);
  lcdg_linha(x1,y1,x0,y1,cor);
  lcdg_linha(x0,y1,x0,y0,cor);
}
void lcdg_retangulo_preenchido(unsigned char x0, unsigned char y0,
            unsigned char x1, unsigned char y1, unsigned char cor)
{
  unsigned char linhas;
  if (y0<y1)
  {
    for (linhas=y0;linhas<=y1;linhas++)
        lcdg_linha(x0,linhas,x1,linhas,cor);
  } else
  {
    for (linhas=y1;linhas<=y0;linhas++)
        lcdg_linha(x0,linhas,x1,linhas,cor);
  }
}
void lcdg_escreve_tela (const unsigned char *matriz)
{
    unsigned char indx,indy;
    unsigned int indz,inicio;
    // configura o apontador de endereços do display para o início da
    // área gráfica
    //lcdg_seta_address_pointer(lcdg_inicio_area_grafica);
    inicio = lcdg_inicio_area_grafica;
    indz=0;  // zero o apontador da matriz
    for (indy=0;indy<lcdg_pixels_y;indy++)
    {
        // configura o apontador de endereços para o início da linha
        lcdg_seta_address_pointer(inicio+indy*lcdg_bytes_por_linha_grafica);
        // seleciona o modo de escrita automática
        lcdg_escreve_comando(lcdg_auto_write_mode);
        // plota uma linha completa da matriz
        for (indx = (lcdg_pixels_x/lcdg_largura_caractere); indx; indx--)
        {
            lcdg_escreve_byte_auto(matriz[indz]);
            indz++;    // incrementa o apontador da matriz
        }
        // desliga o modo automático
        lcdg_escreve_comando(lcdg_desliga_auto_mode);
    }
    // restaura o apontador de endereços do display para o início
    // da área gráfica
    lcdg_seta_address_pointer(lcdg_inicio_area_grafica);
}
```

Exemplo 7-8

O arquivo do exemplo 7-7 deve ser salvo como "lcdg.h" e o arquivo do exemplo 7-8 deve ser salvo como "lcdg.c", ambos na mesma pasta do projeto.

7.3.1. Voltímetro Digital Gráfico

O exemplo seguinte mostra o projeto de um voltímetro digital com apresentação gráfica da tensão lida. Com ele demonstramos a utilização da biblioteca descrita neste tópico, além de demonstrar a utilização de um *display* desse tipo para a plotagem de gráficos. O exemplo foi desenvolvido na placa Microlab X1 utilizando como fonte analógica a entrada ANA2, conectada a um *trimpot* na própria placa. Utilizamos o módulo de CPU MSP430F149, mas o programa pode ser modificado facilmente para utilizar outras CPUs, inclusive da família 44x.

O esquema elétrico de conexão do módulo LCD pode ser visto na figura 7-6, na qual também está o circuito do inversor de tensão e o circuito de acionamento da iluminação de fundo do *display*. **Lembre-se de que os pinos de dados do *display* (DB7 a DB0) devem ser conectados a resistores de 2K2 e estes conectados à porta da CPU (no exemplo foi utilizada a porta P1).**

O barramento de dados deve ser conectado à porta 1, a linha LCD_CE conectada ao pino P2.0, a linha LCD_CD conectada ao pino P2.1, a linha WRITE conectada ao pino P2.2 e a linha READ conectada ao pino P2.3. A linha LCD_PWM (do inversor de tensão) deve ser conectada ao pino P4.6. A linha LCD_FS é utilizada para selecionar o tipo de fonte (quando em nível "0" seleciona a fonte 6x8 e quando em "1", seleciona a fonte 8x8) e a linha LCD_RV é utilizada para selecionar o modo reverso (nível "0") ou normal (nível "1").

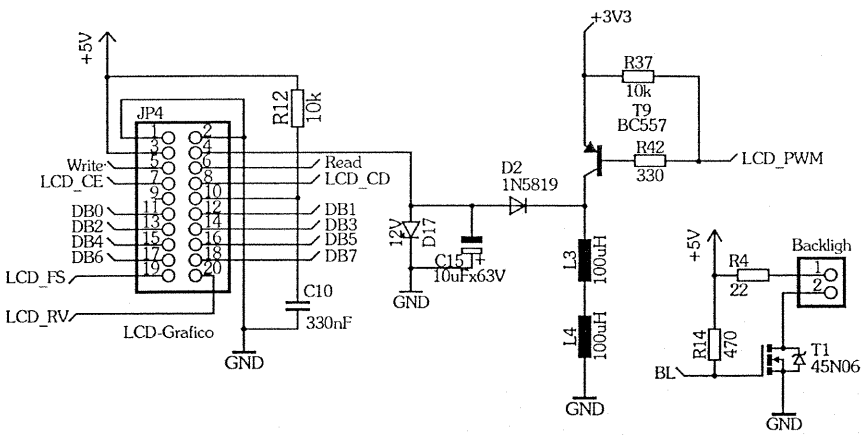


Figura 7-6

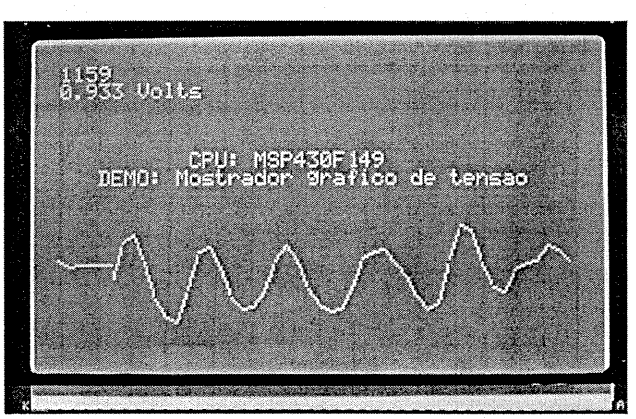


Figura 7-7

```
//*****************************************************************************
// Voltímetro Gráfico
//*****************************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este programa implementa um voltímetro digital que apresenta graficamente
// a tensão em um display LCD. Foi utilizado um LCD gráfico modelo
// G241281WNHDWB, que utiliza um controlador T6963C, um MSP430F149 e a placa
// Microlab X1
//*****************************************************************************
#include <io430x14x.h>
#include <stdio.h>
#include "lcdg.h"
#include "lcdg.c"   // caso se utilize a biblioteca compilada esta linha
                    // pode ser removida
void main(void)
{
   unsigned char posicao=0;
   unsigned int ultimo,atual;
   unsigned long int temp;
   unsigned char string[10];
   WDTCTL = WDTPW + WDTHOLD;
   // para a família 44x:
   // configura o FLL para 8MHz (a partir do LFXT1 a 32768Hz
   //SCFI0 = FLLD_2 + FN_3;
   //SCFQCTL = 0x79;
   //FLL_CTL0 = DCOPLUS;
   //FLL_CTL1 = 0;
   // configura o DCO para a frequência máxima (aprox. 5MHz)
   DCOCTL = 0xFF;
   BCSCTL1 = RSEL2+RSEL1+RSEL0;
   // configura o ADC12 para medição contínua da memória 0
   // no primeiro canal de entrada
   ADC12CTL1 = CONSEQ_2 + SHP;
   ADC12MCTL0 = 0;
   ADC12CTL0 = MSC + ADC12ON + ENC + ADC12SC;
   P6DIR_bit.P6DIR_0 = 0;    // pino P6.0 como entrada
   P6SEL_bit.P6SEL_0 = 1;    // pino P6.0 na função ADC
   // as linhas a seguir configuram o pino 4.6 para o modo PWM,
   // operando em uma frequência de aproximadamente 19,8KHz e um
   // ciclo inativo = 25%. Este sinal é aplicado ao circuito inversor
   // de tensão da placa Microlab X1, de forma a gerar a tensão de -12V
   // para o contraste do display. O inversor é ativado na fase inativa
   // do sinal de PWM
   TBCTL = TBSSEL_2 + MC_1;
   TBCCTL6 = OUTMOD_2;       // o PWM utiliza o canal 6 do timer B
   TBCCR0 = 0xFF;   // período do sinal
   TBCCR6 = 0x61;   // ciclo inativo
   // configura o pino 4.6 para a função de saída do canal 6 do timer B
   // na família 44x, o pino de saída deve ser o P3.7
   P4DIR_bit.P4DIR_6 = 1;
   P4SEL_bit.P4SEL_6 = 1;
   //P3DIR_bit.P3DIR_7 = 1; // para a família 44x
   //P3SEL_bit.P3SEL_7 = 1; // para a família 44x
   lcdg_ini();      // inicializa o display gráfico
   ultimo = 0;
   lcdg_apaga_tela_texto(); // apaga a tela de texto
   lcdg_apaga_tela_grafica();        // apaga a tela gráfica
   lcdg_posiciona_texto(10,5);
   lcdg_escreve_string("CPU: MSP430F149");
   lcdg_posiciona_texto(3,6);
   lcdg_escreve_string("DEMO: Mostrador grafico de tensao");
   lcdg_seta_modo(lcdg_modo_XOR);    // seleciona modo XOR
   while(1)
   {
      for (posicao=0;posicao<=230;posicao=posicao+5)
      {
         // apaga a área da tela antes de plotar
         lcdg_retangulo_preenchido(posicao,0,posicao+5,127,0);
         // lê e ajusta o resultado da conversão
         atual = (long)ADC12MEM0 * 127 / 4095;
         // plota a linha
         lcdg_linha(posicao,ultimo,posicao+5,127-atual,1);
         ultimo = 127-atual;        // guarda a última posição
```

```
            lcdg_posiciona_texto(0,0);
            sprintf(string,"%4d",ADC12MEM0);
            lcdg_escreve_string(string);        // escreve o valor da conversão
            lcdg_posiciona_texto(0,1);
            temp = (long)ADC12MEM0 * 3300 / 4095;
            sprintf(string,"%04ld",temp);
            string[5]=0;
            string[4]=string[3];
            string[3]=string[2];
            string[2]=string[1];
            string[1]='.';
            lcdg_escreve_string(string);        // escreve o valor da tensão
            lcdg_escreve_string(" Volts");
        }
    }
}
```

Exemplo 7-9

7.4. Varredura de Teclado

Uma outra facilidade sempre necessária em uma aplicação microcontrolada é um teclado. Neste exemplo veremos como efetuar a leitura de um teclado composto por doze teclas dispostas em uma matriz e multiplexadas em quatro linhas e três colunas.

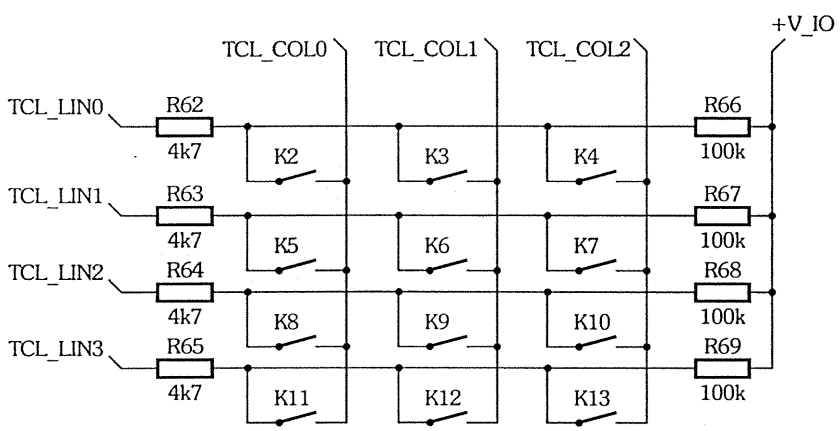


Figura 7-8

Como podemos ver na figura 7-8, o teclado é acessado por sete conexões: TCL_LIN0 a TCL_LIN3 para as linhas e TCL_COL0 a TCL_COL2 para as colunas.

A leitura do teclado é feita colocando um nível lógico "0" em cada sinal de ativação de coluna e monitorando os sinais TCL_LINx. Caso uma tecla daquela coluna esteja pressionada, a respectiva linha terá um nível lógico "0"; caso contrário, a linha apresenta um nível "1". Essa operação é repetida para cada coluna até que todas as colunas tenham sido testadas.

O exemplo seguinte apresenta uma função bastante simples para efetuar a varredura do teclado da figura 7-8. Ela implementa um sistema de filtragem de ruído (debounce) bastante rudimentar, mas que atende aos propósitos didáticos deste livro.

O arquivo deve ser salvo como "teclado3x4.c" na mesma pasta dos projetos em que for utilizado.

```
//*********************************************************************
// Varredura de teclado 3 x 4
//*********************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*********************************************************************
// Função de tratamento do teclado 3x4 da placa Microlab X1. As fun-
// ções neste arquivo possuem o caráter didático e não foram otimiza-
// das para aplicações reais.
//*********************************************************************
// As definições a seguir são utilizadas para acesso aos pinos do
// módulo LCD gráfico
// Para redefinir estes pinos, basta declará-los no programa, antes
// da inclusão desta biblioteca
#ifndef tcl_col0
   #define tcl_col0 P3OUT_bit.P3OUT_0
   #define tcl_col0_dir P3DIR_bit.P3DIR_0
   #define tcl_col1 P3OUT_bit.P3OUT_1
   #define tcl_col1_dir P3DIR_bit.P3DIR_1
   #define tcl_col2 P3OUT_bit.P3OUT_2
   #define tcl_col2_dir P3DIR_bit.P3DIR_2
   #define tcl_lin0 P3IN_bit.P3IN_3
   #define tcl_lin0_dir P3DIR_bit.P3DIR_3
   #define tcl_lin1 P3IN_bit.P3IN_4
   #define tcl_lin1_dir P3DIR_bit.P3DIR_4
   #define tcl_lin2 P3IN_bit.P3IN_5
   #define tcl_lin2_dir P3DIR_bit.P3DIR_5
   #define tcl_lin3 P3IN_bit.P3IN_6
   #define tcl_lin3_dir P3DIR_bit.P3DIR_6
#endif
#pragma inline=forced
void teclado_coluna0(void)
{
  tcl_col0 = 0;  tcl_col1 = 1;  tcl_col2 = 1;
}
#pragma inline=forced
void teclado_coluna1(void)
{
  tcl_col0 = 1;  tcl_col1 = 0;  tcl_col2 = 1;
}
#pragma inline=forced
void teclado_coluna2(void)
{
  tcl_col0 = 1;  tcl_col1 = 1;  tcl_col2 = 0;
}
// função de atraso muito simples
void tcl_delay(unsigned int tempo)
{
  unsigned int temp;
  for (temp = tempo; temp; temp--);
}
//*********************************************************************
// Função varredura do teclado
//*********************************************************************
// Argumentos de chamada: void
//*********************************************************************
// Retorno : unsigned char - valor da tecla pressionada (255 para nenhuma)
//*********************************************************************
unsigned char varre_teclado(void)
{
   static unsigned char ultima;
   unsigned char tecla;
   // configura as portas utilizadas no teclado
   // colunas são saídas
   tcl_col0_dir = 1;
   tcl_col1_dir = 1;
   tcl_col2_dir = 1;
   // linhas são entradas
   tcl_lin0_dir = 0;
   tcl_lin1_dir = 0;
   tcl_lin2_dir = 0;
   tcl_lin3_dir = 0;
   tecla=255;
```

```
    teclado_coluna0();          // ativa a coluna 0
    tcl_delay(100);             // aguarda um tempo
    // verifica as linhas
    if (!tcl_lin0) tecla=1;
    if (!tcl_lin1) tecla=4;
    if (!tcl_lin2) tecla=7;
    if (!tcl_lin3) tecla=10;
    teclado_coluna1();          // ativa a coluna 1
    tcl_delay(100);             // aguarda um tempo
    // verifica as linhas
    if (!tcl_lin0) tecla=2;
    if (!tcl_lin1) tecla=5;
    if (!tcl_lin2) tecla=8;
    if (!tcl_lin3) tecla=0;
    teclado_coluna2();          // ativa a coluna 2
    tcl_delay(100);             // aguarda um tempo
    // verifica as linhas
    if (!tcl_lin0) tecla=3;
    if (!tcl_lin1) tecla=6;
    if (!tcl_lin2) tecla=9;
    if (!tcl_lin3) tecla=11;
    // se alguma tecla foi ou está pressionada
    if (tecla<=11)
    {
       // se a tecla atual for diferente da última
       if (tecla!=ultima)
       {
          ultima = tecla;       // memoriza a tecla atual
          return (tecla);       // e retorna o valor da tecla
       } else
       {
          // se for a mesma tecla que ainda está pressionada
          ultima = tecla;       // memoriza a tecla
          return (255);         // retorna 255
       }
    }
    // se nenhuma tecla está pressionada
    ultima = 255;       // memoriza
    return (255);       // retorna 255
}
```

Exemplo 7-10

7.5. Controlando um Servo com PWM

O exemplo mostra como controlar um servo do tipo utilizado em aeromodelismo e automodelismo R/C. Esses servos são confeccionados por uma grande quantidade de empresas, destacando-se a Futaba, Hobbico, JR, Hitec, entre outras.

A placa Microlab X1 utilizada para escrever este livro possui seis saídas/entradas para servos compatíveis com os citados anteriormente. No exemplo seguinte, foram utilizados servos Hobbico modelo CS-35MG, no entanto outros modelos genéricos e compatíveis podem ser utilizados sem problemas, respeitando a capacidade de suprimento de corrente da placa.

O programa mantém um servo em constante movimento lento da esquerda para direita e no sentido reverso. Um segundo servo pode ser controlado pelo teclado da placa. Cada tecla corresponde a uma posição predefinida do servo.

```
//*********************************************************************
// Controle de servos R/C por PWM
//*********************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*********************************************************************
// Este programa demonstra a utilização do timer B dos MSP430 na
// geração de sinais de PWM. No presente caso, os sinais gerados
// pelos canais 4 e 5 são utilizados para o controle de dois servos
// R/C conforme descrito no texto do livro. Foi utilizada a placa
// Microlab X1, um módulo MSP430F149 e dois servos Hobbico modelo
// CS-35MG.
//*********************************************************************

#include <io430x14x.h>
#include "lcdc.h"
#include "lcdc.c"   // caso se utilize a biblioteca compilada esta linha
                    // pode ser removida
#include "teclado3x4.c"  // função de leitura do teclado

void main(void)
{
   unsigned char tecla,direcao,temporizacao = 100;
   unsigned int servo_pos;
   WDTCTL = WDTPW + WDTHOLD; // desativa o watchdog
   // clock interno, aproximadamente 5MHz
   DCOCTL = 255;
   BCSCTL1 = 7;
   // Configura os canais 4 e 5 do timer B para o modo PWM
   // A freqüência do sinal é de aproximadamente 82Hz
   // a variação do ciclo ativo permite alterar a duração dos
   // pulsos na faixa de operação dos servos, que é de 1 a 2ms
   TBCTL = MC_1 + TBSSEL_2;
   TBCCTL4 = OUTMOD_2;
   TBCCTL5 = OUTMOD_2;
   TBCCR0 = 0xf000;      // período do sinal
   TBCCR5 = 0xcd00;      // ciclo ativo do canal 5
   TBCCR4 = 0xcd00;      // ciclo ativo do canal 4
   // configura os pinos P4.4 e P4.5 para a função de saída
   // do timer B
   P4SEL_bit.P4SEL_4=1;
   P4DIR_bit.P4DIR_4=1;
   P4SEL_bit.P4SEL_5=1;
   P4DIR_bit.P4DIR_5=1;
   // configura o pino P2.2 como saída e o coloca em nível "0"
   // o pino write do módulo deve ser conectado ao mesmo
   P2DIR_bit.P2DIR_2 = 1;
   P2OUT_bit.P2OUT_2 = 0;
   // inicializa o módulo LCD
   lcdc_ini(lcdc_display_8x5|lcdc_2_linhas,lcdc_display_ligado|
         lcdc_cursor_desligado|lcdc_cursor_fixo);
   // apaga a tela
   lcdc_escreve_char ('\f');
   lcdc_posiciona_texto(3,0);
   lcdc_escreve_string("MSP430F149");
   lcdc_posiciona_texto(0,1);
   servo_pos = 0xbe00;   // posição inicial do servo
   direcao = 1;          // direção de deslocamento
   while(1)
   {
      tecla=varre_teclado();         // verifica o teclado
      if (temporizacao) temporizacao--; else
      {
         // movimenta automaticamente o servo conectado ao canal 5
         if (direcao) servo_pos += 10; else servo_pos -= 10;
         if (servo_pos>0xe200) direcao = 0;
         if (servo_pos<0xbf00) direcao = 1;
         TBCCR5 = servo_pos;
         temporizacao = 100;
      }
      if (tecla==10)     // se for pressionada a tecla *, apaga a tela
      {
         lcdc_escreve_char('\f');
         lcdc_posiciona_texto(3,0);
         lcdc_escreve_string("MSP430F149");
```

```
            lcdc_posiciona_texto(0,1);
        }
        if (tecla<=9) // se a tecla for menor igual a 9
        {
            // escreve o valor da tecla no display
            lcdc_escreve_char(tecla+'0');
            switch (tecla)
            {
                // posiciona o servo ligado ao canal 4, na posição selecionada
                case 0 :    TBCCR4 = 0xbe00;
                            break;
                case 1 :    TBCCR4 = 0xc200;
                            break;
                case 2 :    TBCCR4 = 0xc600;
                            break;
                case 3 :    TBCCR4 = 0xca00;
                            break;
                case 4 :    TBCCR4 = 0xce00;
                            break;
                case 5 :    TBCCR4 = 0xd200;
                            break;
                case 6 :    TBCCR4 = 0xd600;
                            break;
                case 7 :    TBCCR4 = 0xda00;
                            break;
                case 8 :    TBCCR4 = 0xde00;
                            break;
                case 9 :    TBCCR4 = 0xe200;
                            break;
            }
        }
    }
}
```

Exemplo 7-11

7.6. Comunicação Serial Assíncrona

O próximo exemplo demonstra a utilização da USART no modo assíncrono para a comunicação com um computador PC.

O programa funciona como um terminal serial muito simples: ao pressionar uma tecla da placa Microlab X1, ela é apresentada no módulo LCD e enviada ao PC. Ao pressionar uma tecla no PC, é recebida pela aplicação e apresentada na tela do módulo LCD.

Foi utilizada a interrupção de alteração do estado da porta 2 para a detecção da tecla pressionada. Além disso, o programa utiliza *buffers* circulares tanto para a recepção quanto para a transmissão. Esses *buffers* permitem que o programa acesse a USART de forma transparente, simplesmente escrevendo e lendo caracteres por meio das funções "X1_RS232cGetChar" e "X1_RS232TXChar".

O circuito de comunicação utiliza um MAX232, como ilustrado na figura 1-5. A linha RX deve ser conectada ao pino P3.5 e a linha TX deve ser conectada ao pino P3.4. A conexão do teclado e do módulo LCD é idêntica aos exemplos anteriores.

Para comunicação com a aplicação, sugerimos o programa Tera Term, uma aplicação de terminal serial compatível com VT100 e que possui distribuição e uso gratuitos (item 46 nas referências bibliográficas).

```
//*****************************************************************************
// Demonstração de terminal serial
//*****************************************************************************
// Autor: Dado Sutter
// Modificações para o livro: Fábio Pereira
//*****************************************************************************
// Este programa demonstra a comunicação entre o MSP430 e um microcompu-
// tador PC. O MSP atua como um terminal serial, lendo o teclado e envian-
// do informações para o PC. Além disso, as informações recebidas do PC
// são apresentadas na tela do módulo LCD. Foi utilizada a placa Microlab
// X1 e um módulo de CPU MSP430F149.
//*****************************************************************************
#include <io430x14x.h>
#include <intrinsics.h>
#include "lcdc.h"
#include "lcdc.c"
// Definições das Portas e Bits utilizados para acesso ao teclado
#define TCL_lin0      P2IN_bit.P2IN_4
#define TCL_lin1      P2IN_bit.P2IN_5
#define TCL_lin2      P2IN_bit.P2IN_6
#define TCL_lin3      P2IN_bit.P2IN_7
#define TCL_col0      P3OUT_bit.P3OUT_0
#define TCL_col1      P3OUT_bit.P3OUT_1
#define TCL_col2      P3OUT_bit.P3OUT_2
// Definições para direção dos bits utilizados para linhas e colunas do
// teclado
#define TCL_lin0_DIR  P2DIR_bit.P2DIR_4
#define TCL_lin1_DIR  P2DIR_bit.P2DIR_5
#define TCL_lin2_DIR  P2DIR_bit.P2DIR_6
#define TCL_lin3_DIR  P2DIR_bit.P2DIR_7
#define TCL_col0_DIR  P3DIR_bit.P3DIR_0
#define TCL_col1_DIR  P3DIR_bit.P3DIR_1
#define TCL_col2_DIR  P3DIR_bit.P3DIR_2
// Definições para a direção das interrupções sendo geradas nas linhas
#define TCL_lin0_IES  P2IES_bit.P2IES_4
#define TCL_lin1_IES  P2IES_bit.P2IES_5
#define TCL_lin2_IES  P2IES_bit.P2IES_6
#define TCL_lin3_IES  P2IES_bit.P2IES_7
// Definições para a habilitação das interrupções nas linhas
#define TCL_lin0_IE   P2IE_bit.P2IE_4
#define TCL_lin1_IE   P2IE_bit.P2IE_5
#define TCL_lin2_IE   P2IE_bit.P2IE_6
#define TCL_lin3_IE   P2IE_bit.P2IE_7
// Definições dos bits individuais das linhas
#define TCL_lin0_IFG  0x10
#define TCL_lin1_IFG  0x20
#define TCL_lin2_IFG  0x40
#define TCL_lin3_IFG  0x80
// Definição dos flags de interrupção gerados pelas linhas
#define TCL_IFG P2IFG
#define KBD_VECTOR PORT2_VECTOR
// Máscara utilizada para a leitura das linhas do teclado
#define TCL_IFG_mask (TCL_lin0_IFG | TCL_lin1_IFG | TCL_lin2_IFG | TCL_lin3_IFG)
//////////////////////////////////////////////////////////////////////////////
// Macros para ativação das colunas
//////////////////////////////////////////////////////////////////////////////
#define TCL_Col0        TCL_col0 = 0; TCL_col1 = 1; TCL_col2 = 1
#define TCL_Col1        TCL_col0 = 1; TCL_col1 = 0; TCL_col2 = 1
#define TCL_Col2        TCL_col0 = 1; TCL_col1 = 1; TCL_col2 = 0
#define TCL_ColNone     TCL_col0 = 1; TCL_col1 = 1; TCL_col2 = 1
#define TCL_ColAll      TCL_col0 = 0; TCL_col1 = 0; TCL_col2 = 0
//////////////////////////////////////////////////////////////////////////////
// Variáveis globais que serão alteradas pelo presente método de leitura do TCL
//////////////////////////////////////////////////////////////////////////////
volatile unsigned char cTecla;              // Código da última tecla
volatile unsigned char bAlgoFoiTeclado;     //Flag Alguma tecla foi teclada
//////////////////////////////////////////////////////////////////////////////
// Inicialização das portas/bits utilizados na varredura do teclado
// Bits das Colunas como saídas, Bits das linhas como entradas
// Após a configuração das portas, ativa 0 lógico nos bits das colunas
// para que as teclas pressionadas gerem interrupções nos bits das linhas
//////////////////////////////////////////////////////////////////////////////
void X1_TCLInit(void)
{
```

```
    // Configura direção dos bits utilizados no teclado
    TCL_lin0_DIR = 0;         // Linhas como entradas
    TCL_lin1_DIR = 0;
    TCL_lin2_DIR = 0;
    TCL_lin3_DIR = 0;
    TCL_col0_DIR = 1;         // Colunas como saídas
    TCL_col1_DIR = 1;
    TCL_col2_DIR = 1;
    // Configura interrupções sendo geradas por transições de nível 1 para 0
    TCL_lin0_IES = 1;
    TCL_lin1_IES = 1;
    TCL_lin2_IES = 1;
    TCL_lin3_IES = 1;
    // Habilita interrupções nos bits relativos às linhas do teclado
    TCL_lin0_IE = 1;
    TCL_lin1_IE = 1;
    TCL_lin2_IE = 1;
    TCL_lin3_IE = 1;
    TCL_ColAll;               // ZERO lógico (GND) nas linhas das colunas
    TCL_IFG = 0;              // Limpa eventuais interrupções pendentes
    bAlgoFoiTeclado = 0;
}
////////////////////////////////////////////////////////////////////////////
//  Rotina de Tratamento de Interrupções da Porta 1
//  Ativada quando uma tecla é pressionada no TCL da Microlab X1
//  Uma vez identificada a linha, varre as colunas para identificar a
//  tecla. Códigos gerados:
//          1   2   3
//          4   5   6
//          7   8   9
//          10  0   11
////////////////////////////////////////////////////////////////////////////
#pragma vector = PORT2_VECTOR
__interrupt void TCLPortISR(void)
{
  unsigned char cLinhaPressionada, i;
  cLinhaPressionada = TCL_IFG & TCL_IFG_mask ;        // máscara de bits
  for (i=255; i; --i);                // um pequeno atraso
  // verifica se a tecla ainda está pressionada
  if (cLinhaPressionada == (TCL_IFG & TCL_IFG_mask))
  {
    switch (cLinhaPressionada)        // varre o teclado
    {
      case TCL_lin0_IFG: TCL_Col0;    // linha 1
                         if (!TCL_lin0) cTecla=1;
                         TCL_Col1;
                         if (!TCL_lin0) cTecla=2;
                         TCL_Col2;
                         if (!TCL_lin0) cTecla=3;
                         break;
      case TCL_lin1_IFG: TCL_Col0;    // linha 2
                         if (!TCL_lin1) cTecla=4;
                         TCL_Col1;
                         if (!TCL_lin1) cTecla=5;
                         TCL_Col2;
                         if (!TCL_lin1) cTecla=6;
                         break;
      case TCL_lin2_IFG: TCL_Col0;    // linha 3
                         if (!TCL_lin2) cTecla=7;
                         TCL_Col1;
                         if (!TCL_lin2) cTecla=8;
                         TCL_Col2;
                         if (!TCL_lin2) cTecla=9;
                         break;
      case TCL_lin3_IFG: TCL_Col0;    // linha 4
                         if (!TCL_lin3) cTecla=10;
                         TCL_Col1;
                         if (!TCL_lin3) cTecla=0;
                         TCL_Col2;
                         if (!TCL_lin3) cTecla=11;
                         break;
      default: cTecla = 0xFF;
    }
    TCL_ColAll;             // ZERO lógico (GND) nas linhas das colunas
```

```
      TCL_IFG = 0x00;        // Limpa eventuais interrupções pendentes
      bAlgoFoiTeclado = 1;    // seta o indicador de tecla pressionada
      // a linha a seguir não é necessária no presente caso, mas poderia
      // ser utilizada caso a CPU estivesse em um modo de baixo consumo
      // quando a tecla foi pressionada
      __low_power_mode_off_on_exit();// Retorna mantendo a CPU no modo Ativo
   }
}
//////////////////////////////////////////////////////////////////////////
// Definições utilizadas pelas rotinas de comunicação serial e USART
//////////////////////////////////////////////////////////////////////////
// Macros para manipulação da habilitação da USART
#define UART_TX_ENABLE         ME1_bit.UTXE0 = 1
#define UART_RX_ENABLE         ME1_bit.URXE0 = 1
#define UART_TX_DISABLE        ME1_bit.UTXE0 = 0
#define UART_RX_DISABLE        ME1_bit.URXE0 = 0
// Macros para controle das interrupções da USART
#define UART_TX_INT_ENABLE     IE1_bit.UTXIE0 = 1
#define UART_TX_INT_DISABLE    IE1_bit.UTXIE0 = 0
#define UART_RX_INT_ENABLE     IE1_bit.URXIE0 = 1
#define UART_RX_INT_DISABLE    IE1_bit.URXIE0 = 0
// Definições do buffer circular de recepção
// tamanho do buffer (deve ser um múltiplo de potência de dois)
#define RXBUFSIZE 32
static volatile unsigned char ucRXBuffer[RXBUFSIZE];
static volatile unsigned char ucRXReadIndex, ucRXWriteIndex;
static volatile unsigned char ucRXCharCount;
// Definições do buffer circular de transmissão
// tamanho do buffer (deve ser um múltiplo de potência de dois)
#define TXBUFSIZE 32
static volatile unsigned char ucTXBuffer[TXBUFSIZE];
static volatile unsigned char ucTXReadIndex, ucTXWriteIndex;
static volatile unsigned char ucTXCharCount;
#define BUFFER_EMPTY 1
static volatile unsigned char bTXBufferEmpty;
//////////////////////////////////////////////////////////////////////////
// Inicialização da biblioteca da USART da Microlab X1
//////////////////////////////////////////////////////////////////////////
void X1_RS232Init(void)
{
   // Inicialização das filas circulares, buffers de Tx e Rx
   ucRXWriteIndex = ucRXReadIndex = ucRXCharCount = 0;
   ucTXWriteIndex = ucTXReadIndex = ucTXCharCount = 0;
   bTXBufferEmpty = BUFFER_EMPTY;  // apaga flag de buffer vazio
   // habilita a USART
   UART_TX_ENABLE;
   UART_RX_ENABLE;
   // Configura os pinos
   P3SEL_bit.P3SEL_5 = 1;    // P3.5 como URXD0
   P3SEL_bit.P3SEL_4 = 1;    // P3.4 como UTXD0
   // configura a USART
   U0CTL  = SWRST + CHAR;    // dados de 8 bits
   // seleciona o SMCLK como fonte de clock da USART
   // (aproximadamente 823KHz)
   U0TCTL = SSEL0+SSEL1;
   U0MCTL = 0xDD;            // velocidade = 9600bps
   U0BR0 = 0x55;
   U0BR1 = 0x00;
   U0CTL_bit.SWRST = 0;      // sai do modo de reset da USART
   // habilita as interrupções
   UART_TX_INT_ENABLE;
   UART_RX_INT_ENABLE;
}
//////////////////////////////////////////////////////////////////////////
// Transmite um caracter via UART (através do buffer)
//////////////////////////////////////////////////////////////////////////
void X1_RS232TXChar (char cByte)
{
   // escreve o caractere no buffer, na posição atual do índice
   // e incrementa o índice
   ucTXBuffer[ucTXWriteIndex++] = cByte;
   // ajusta o índice de escrita
   ucTXWriteIndex &= TXBUFSIZE-1;
   // desliga a interrupção de transmissão
```

```
    UART_TX_INT_DISABLE;
    // incrementa o contador de caracteres a serem transmitidos
    ucTXCharCount++;
    // habilita a interrupção de transmissão
    UART_TX_INT_ENABLE;
    // se o indicador de buffer vazio estiver setado e ainda
    // houver caracteres no buffer de transmissão
    if (bTXBufferEmpty && ucTXCharCount)
    {
      // apaga o indicador de buffer vazio
     bTXBufferEmpty = !BUFFER_EMPTY;
     // escreve o próximo caractere do buffer na USART e
     // incrementa o índice de leitura do buffer
     U0TXBUF = ucTXBuffer[ucTXReadIndex++];
     // ajusta o índice de leitura
     ucTXReadIndex &= TXBUFSIZE-1;
     // decrementa o número de caracteres a transmitir
     ucTXCharCount--;
    }
}
////////////////////////////////////////////////////////////////////////////
// Tratamento da interrupção de transmissão da USART
////////////////////////////////////////////////////////////////////////////
#pragma vector = UART0TX_VECTOR
__interrupt void TXInterrupt (void)
{
    if (ucTXCharCount)        // se há caracteres no buffer
    {
     // envia o caractere apontado pelo índice e incrementa o índice
     U0TXBUF = ucTXBuffer[ucTXReadIndex++];
     ucTXReadIndex &= TXBUFSIZE-1;    // ajusta o índice
     // decrementa a contagem de caracteres a transmitir
     ucTXCharCount--;
    } else   // se o buffer estiver vazio
    bTXBufferEmpty = BUFFER_EMPTY;    // seta o indicador
}
////////////////////////////////////////////////////////////////////////////
// Lê um caractere da USART (através do buffer)
////////////////////////////////////////////////////////////////////////////
char X1_RS232cGetChar (void)
{
  char Byte;
  if (ucRXCharCount)         // há caracteres no buffer ?
  {
     // Lê o caractere apontado pelo índice e incrementa o índice
    Byte = ucRXBuffer[ucRXReadIndex++];
    // ajusta o índice
    ucRXReadIndex &= RXBUFSIZE-1;
    // desliga a interrupção de recepção
    UART_RX_INT_DISABLE;
    // decrementa o número de caracteres lidos
    ucRXCharCount--;
    // habilita a interrupção de recepção
    UART_RX_INT_ENABLE;
    return (Byte);  // retorna o caractere
  } else return (0);         // retorna zero caso não exista um caractere
}
////////////////////////////////////////////////////////////////////////////
// Tratamento da interrupção de recepção
////////////////////////////////////////////////////////////////////////////
#pragma vector = UART0RX_VECTOR
__interrupt void RXInterrupt (void)
{
  //_EINT();
  // lê o dado recebido e escreve na posição atual do buffer
  // depois incrementa o índice
  ucRXBuffer[ucRXWriteIndex++] = U0RXBUF;
  // ajusta o índice
  ucRXWriteIndex &= RXBUFSIZE-1;
  // incrementa a contagem de caracteres recebidos
  ucRXCharCount++;
}
////////////////////////////////////////////////////////////////////////////
// Retorna o quantidade de caracteres no buffer de recepção
```

```
//////////////////////////////////////////////////////////////////////////
unsigned char X1_ucRS232RXBufferCount (void)
{
  return (ucRXCharCount);
}
//////////////////////////////////////////////////////////////////////////
// Rotina Principal
//////////////////////////////////////////////////////////////////////////
void main(void)
{
   WDTCTL = WDTPW + WDTHOLD;          // Desabilita Watchdog Timer
   X1_TCLInit();                              // inicializa o teclado
   X1_RS232Init();                    // inicializa a USART
   bAlgoFoiTeclado = 0;      // inicializa Flag
   // configura o pino P2.2 como saída e o coloca em nível "0"
   // o pino write do módulo deve ser conectado ao mesmo
   P2DIR_bit.P2DIR_2 = 1;
   P2OUT_bit.P2OUT_2 = 0;
   // inicializa o módulo LCD
   lcdc_ini(lcdc_display_8x5|lcdc_2_linhas,lcdc_display_ligado|
            lcdc_cursor_desligado|lcdc_cursor_fixo);
   lcdc_escreve_char ('\f');          // apaga a tela
   lcdc_posiciona_texto(3,0);
   lcdc_escreve_string("MSP430F149");
   lcdc_posiciona_texto(0,1);
  __enable_interrupt();      // habilita as interrupções
  while(1)
  {


    // verifica se há um caractere no buffer
     if (X1_ucRS232RXBufferCount())
     {
        // se existe, escreve-o no display
        lcdc_escreve_char(X1_RS232cGetChar());
     }
     if (bAlgoFoiTeclado)    // uma tecla foi pressionada ?
     {
        bAlgoFoiTeclado = 0;// apaga o indicador
        if (cTecla < 10)              // se a tecla é um número
        {
           // escreve o valor da cTecla no display
           lcdc_escreve_char(cTecla + '0');
           X1_RS232TXChar(cTecla + '0');     // ASCII Zero + offset da cTecla
        } else
        if (cTecla == 10)
        {
           // se a tecla for a K11
           lcdc_escreve_char('\f'); // apaga a tela do LCD
           lcdc_posiciona_texto(3,0);
           lcdc_escreve_string("MSP430F149");
           lcdc_posiciona_texto(0,1);
           X1_RS232TXChar('*');      // envia *
        } else
        // se a tecla for a K13
        if (cTecla == 11) X1_RS232TXChar(0x13);       // envia ENTER
     }
  }
}
```

Exemplo 7-12

7.7. Comunicação SPI

No próximo exemplo, demonstramos a comunicação do MSP430 com um CI HT1381. Trata-se de um RTC (***Real Time Clock*** - Relógio de Tempo Real) fabricado pela Holtek, capaz de manter a contagem de tempo e calendário, consumindo uma corrente muito baixa e livrando a CPU desse tipo de tarefa.

É claro que um microcontrolador voltado para aplicações de baixo consumo, como o MSP430, a princípio, não necessita desse tipo de periférico, mas resolvemos incluir este exemplo para demonstrar as potencialidades da interface SPI dos MSP430, além de demonstrar também uma técnica SPI que utiliza uma linha de dados bidirecional.

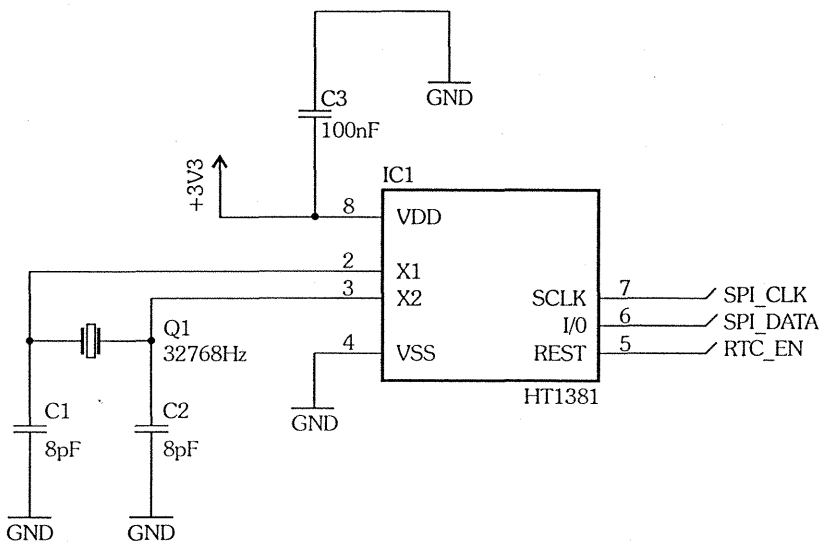


Figura 7-9

A configuração e utilização do HT1381 (ou da sua versão DIP, o HT1380) é muito simples. A tabela seguinte apresenta os registradores internos e os comandos de acesso, que devem ser transmitidos serialmente pela SPI. Primeiramente se transmite o comando e em seguida, o dado pode ser lido ou escrito, conforme o caso.

O *bit* WP protege os registradores do *chip* contra escrita e deve ser programado em nível "0" antes de poder configurar o restante do *chip*.

O *bit* CH controla o funcionamento do oscilador e deve ser colocado em "0" para que ele opere. Quando esse *bit* está em nível "1", o *chip* entra em modo de espera, consumindo cerca de 100nA. No modo ativo o consumo é da ordem de 1μA.

O *bit* 12/24 controla o modo de operação do relógio: em "1" temos o modo 12 horas e em "0", o modo 24 horas. Quando operando no modo 12 horas, o *bit* AP pode ser utilizado para saber se o horário é de antes do meio-dia (AM) quando AP=0 ou depois do meio-dia (PM), quando AP=1.

Registrador	Faixa de dados	Definição dos *bits* do registrador								Endereço A2 - A0	*Bit* R/W	Comando
		Bit 7	*Bit* 6	*Bit* 5	*Bit* 4	*Bit* 3	*Bit* 2	*Bit* 1	*Bit* 0			
Segundos	00-59	CH	Dezenas de segundos			Segundos				000	Escrita	10000000
											Leitura	10000001
Minutos	00-59	0	Dezenas de minutos			Minutos				001	Escrita	10000010
											Leitura	10000011
Hora	01-12	12/	0	AP	HR	Horas				010	Escrita	10000100
	00-23	24		10	HR						Leitura	10000101
Dia do mês	01-31	0	0	10D		Dia do mês				011	Escrita	10000110
											Leitura	10000111
Mês	01-12	0	0	0	10M	Mês				100	Escrita	10001000
											Leitura	10001001
Dia da semana	01-07	0	0	0	0	Dia da semana				101	Escrita	10001010
											Leitura	10001011
Ano	00-99	Dezenas do ano				Ano				110	Escrita	10001100
											Leitura	10001101
Proteção contra escrita	00-80	WP	0	0	0	0	0	0	0	111	Escrita	10001110
											Leitura	10001111

Tabela 7-4

Os registradores armazenam o horário e datas utilizando o formato BCD, o que facilita a sua manipulação.

Há também um modo de leitura e escrita em rajadas (*burst*) que permite que se leia ou escreva em todos os registradores em seqüência (com exceção do *bit* WP que não pode ser escrito nesse modo). Para realizar uma leitura em rajada, utiliza-se o comando 10111111 (0xBF) e para escrita, utiliza-se o comando 10111110 (0xBE). Após o envio do comando, seguem-se 8 *bytes* com o conteúdo dos registradores, iniciando pelo *bit* 0 do registrador 0 (segundos).

Vale ressaltar que na comunicação com o HT1381 o primeiro *bit* transmitido é o LSB e o último o MSB, o que difere do padrão da USART dos MSP430 que é iniciado pelo MSB e terminado pelo LSB, por isso foi incluída uma pequena função para inverter a ordem dos *bits* de uma variável de 8 *bits*.

O exemplo seguinte utiliza um MSP430F149 com a linha SPI_CLK conectada ao pino P3.3 e a linha SPI_DATA conectada aos pinos P3.2 e P3.1. A linha de seleção do chip (RTC_EN) foi conectada ao pino P3.7.

```
//*****************************************************************************
// Demonstração do uso do HT1381
//*****************************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este programa demonstra a comunicação com o chip HT1381 utilizando a
// USART0 no modo SPI. Foi utilizada a placa Microlab X1 e um módulo de CPU
// MSP430F149. O HT1381 já está integrado na placa Microlab.
//*****************************************************************************
#include <io430x14x.h>
#include "lcdc.h"
#include "lcdc.c"   // caso se utilize a biblioteca compilada esta linha
                    // pode ser removida
#include <intrinsics.h>
#define spi_data_pin_in P3IN_bit.P3IN_2
#define spi_data_pin_out P3OUT_bit.P3OUT_2
#define spi_data_pin_dir P3DIR_bit.P3DIR_2
#define spi_clk_pin P3OUT_bit.P3OUT_3
```

```
#define spi_clk_pin_dir P3DIR_bit.P3DIR_3
#define rtc_en_pin P3OUT_bit.P3OUT_7
#define rtc_en_pin_dir P3DIR_bit.P3DIR_7
// comandos do HT1381
#define ht138x_escreve_seg 0x80
#define ht138x_escreve_min 0x82
#define ht138x_escreve_hora 0x84
#define ht138x_escreve_data 0x86
#define ht138x_escreve_mes 0x88
#define ht138x_escreve_dia 0x8A
#define ht138x_escreve_ano 0x8C
#define ht138x_escreve_wp 0x8E
#define ht138x_le_seg 0x81
#define ht138x_le_min 0x83
#define ht138x_le_hora 0x85
#define ht138x_le_data 0x87
#define ht138x_le_mes 0x89
#define ht138x_le_dia 0x8B
#define ht138x_le_ano 0x8D
#define ht138x_le_wp 0x8F
#define ht138x_protegido 0x80
#define ht138x_desprotegido 0x00
#define ht138x_wp 0x01      // proteção contra escrita
#define ht138x_ch 0x02      // parada de clock
#define ht138x_24 0x04      // modo 24 horas

struct ehorario
{
   char hora;
   char minuto;
   char segundo;
} horario;

struct edata
{
   char ano;
   char mes;
   char dia;
   char data;
} calendario;

//*****************************************************************************
// usart_spi_inicializa
//*****************************************************************************
// Recebe : void    Retorna: void
//*****************************************************************************
// Esta função inicializa os registradores e pinos da USART para o modo SPI
//*****************************************************************************
void usart_spi_inicializa()
{
   // Configura os pinos da USART para o modo alternativo
   P3SEL_bit.P3SEL_1 = 1;
   P3SEL_bit.P3SEL_2 = 1;
   P3SEL_bit.P3SEL_3 = 1;
   // Configura a USART
   ME1 = URXE0 + UTXE0;      // habilita o módulo
   // Modo = Síncrono, Master, 8 Bits
   U0CTL = SWRST + CHAR + SYNC + MM;
   // Clock = SMCLK, STE desabilitado, polaridade 1
   U0TCTL = SSEL1 + SSEL0 + STC + CKPH;
   U0MCTL = 0;       // modulador = 0
   U0BR0 = 0x2;      // UCLK = SMCLK / 2
   U0BR1 = 0x00;
   U0CTL_bit.SWRST = 0;      // apaga o reset da USART
}

//*****************************************************************************
// espelha_byte
//*****************************************************************************
// Recebe : unsigned char valor - o valor a ser espelhado
// Retorna: unsigned char - o valor espelhado
```

```
//*****************************************************************************
// Esta função faz a inversão da ordem dos bits do dado recebido
//*****************************************************************************
unsigned char espelha_byte(unsigned char valor)
{
   unsigned int val,aux,result;
   aux = 128;
   result = 0;
   for (val=1;val<=128;val*=2)
   {
      if (valor & val) result |= aux;
      aux = aux >> 1;
   }
   return (result);
}

//*****************************************************************************
// usart_spi_escreve
//*****************************************************************************
// Recebe :         unsigned char comando - o comando a ser enviado
//                  unsigned char dado - o dado a ser enviado
// Retorna:         void
//*****************************************************************************
// Transmite um comando seguido de um dado
//*****************************************************************************
void usart_spi_escreve(unsigned char comando, unsigned char dado)
{
   volatile unsigned char temp;
   // configura o pino P3.1 para o modo USART
   P3SEL_bit.P3SEL_1 = 1;
   // envia o comando
   U0TXBUF = espelha_byte(comando);
   while (!U0TCTL_bit.TXEPT);         // aguarda a transmissão
   temp = U0RXBUF;           // lê o dado recebido e descarta
   U0TXBUF = espelha_byte(dado);      // envia o dado
   while (!U0TCTL_bit.TXEPT);         // aguarda a transmissão
   temp = U0RXBUF;           // lê o dado recebido e descarta
}

//*****************************************************************************
// usart_spi_le
//*****************************************************************************
// Recebe :         unsigned char comando - o comando a ser enviado
// Retorna:         unsigned char - o dado lido da SPI
//*****************************************************************************
// Transmite um comando e lê o dado recebido
//*****************************************************************************
unsigned char usart_spi_le(unsigned char comando)
{
   volatile unsigned char temp;
   // configura o pino P3.1 para o modo USART
   P3SEL_bit.P3SEL_1 = 1;
   // envia o comando
   U0TXBUF = espelha_byte(comando);
   while (!U0TCTL_bit.TXEPT);         // aguarda a transmissão
   temp = U0RXBUF;  // lê o dado recebido e descarta
   // configura o pino P3.1 como entrada
   P3SEL_bit.P3SEL_1 = 0;
   P3DIR_bit.P3DIR_1 = 0;
   // envia um comando, apenas para ativar o clock e poder receber
   // o dado
   U0TXBUF = espelha_byte(comando);
   while (!U0TCTL_bit.TXEPT);                // aguarda a transmissão
   return(espelha_byte(U0RXBUF));    // retorna o dado recebido
}
```

```
//*****************************************************************************
// ht138x_escreve_horario
//*****************************************************************************
// Recebe :         struct ehorario horario - estrutura com o horário
// Retorna:         void
//*****************************************************************************
// Escreve um horário no HT138x
//*****************************************************************************
void ht138x_escreve_horario(struct ehorario horario)
{
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   // envia comando para desproteger o chip contra escrita
   spi_escreve(ht138x_escreve_wp,ht138x_desprotegido);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
   __no_operation();         // aguarda alguns ciclos
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   // configura os minutos
   usart_spi_escreve(ht138x_escreve_min,horario.minuto);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
   __no_operation();         // aguarda alguns ciclos
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   // escreve as horas
   usart_spi_escreve(ht138x_escreve_hora,horario.hora);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
   __no_operation();         // aguarda alguns ciclos
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   // escreve novamente a hora
   usart_spi_escreve(ht138x_escreve_hora,horario.hora);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
}
//*****************************************************************************
// ht138x_escreve_horario
//*****************************************************************************
// Recebe :         void
// Retorna:         struct ehorario - estrutura com o horário
//*****************************************************************************
// Escreve um horário no HT138x
//*****************************************************************************
struct ehorario ht138x_le_horario(void)
{
   struct ehorario horario_temp;
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   // lê os minutos
   horario_temp.minuto = usart_spi_le(ht138x_le_min);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
   __no_operation();
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   // lê os minutos
   horario_temp.minuto = usart_spi_le(ht138x_le_min);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
   __no_operation();
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   // lê as horas
   horario_temp.hora = usart_spi_le(ht138x_le_hora);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
   __no_operation();
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   // lê os segundos
   horario_temp.segundo = usart_spi_le(ht138x_le_seg);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
   return (horario_temp);    // retorna a estrutura
}
```

```
//*****************************************************************************
// ht138x_inicializa
//*****************************************************************************
// Recebe :          unsigned char estado - estado a ser configurado
// Retorna:          void
//*****************************************************************************
// Configura o HT138x. Podemos utilizar os seguintes símbolos para a
// configuração:
// ht138x_wp - para ativar a proteção contra escrita
// ht138x_ch - para paralisar o clock e colocar o chip em standby
// ht138x_24 - para selecionar o modo 24 horas
// Após a configuração o horário é apagado
//*****************************************************************************
void ht138x_inicializa(unsigned char estado)
{
   char temp;
   rtc_en_pin_dir = 1;       // configura o pino RTC_EN como saída
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   if (estado && ht138x_wp) temp = 0x80; else temp = 0x00;
   usart_spi_escreve(ht138x_escreve_wp,temp);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   if (estado && ht138x_ch) temp = 0x80; else temp = 0x00;
   usart_spi_escreve(ht138x_escreve_seg,temp);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
   rtc_en_pin = 1;  // coloca o pino RTC_EN em 1
   if (estado && ht138x_24) temp = 0x00; else temp = 0x80;
   usart_spi_escreve(ht138x_escreve_hora,temp);
   rtc_en_pin = 0;  // desativa a linha RTC_EN
}

void main( void )
{
   unsigned char digito;
   WDTCTL = WDTPW + WDTHOLD; // desativa o watchdog
   // configura o horário para 09:59:59
   horario.hora = 0x09;
   horario.minuto = 0x59;
   horario.segundo = 0x59;
   usart_spi_ini(); // inicializa a USART para o modo SPI
   ht138x_ini(0);   // inicializa o HT1381
   ht138x_escreve_horario(horario);
   // configura o pino P2.2 como saída e o coloca em nível "0"
   // o pino write do módulo deve ser conectado ao mesmo
   P2DIR_bit.P2DIR_2 = 1;
   P2OUT_bit.P2OUT_2 = 0;
   lcdc_ini(lcdc_display_8x5|lcdc_2_linhas,lcdc_display_ligado|lcdc_cursor_desligado|lcdc
_cursor_fixo);
   lcdc_escreve_char ('\f'); // apaga a tela
   lcdc_posiciona_texto(2,0);
   lcdc_escreve_string("Horario");
   while (1)
   {
      horario = ht138x_le_horario();         // lê o horário do HT1381
      lcdc_posiciona_texto(2,1);
      digito = (horario.hora>>4)+'0';                 // separa o MSD da hora
      lcdc_escreve_char(digito);                      // escreve o dígito
      digito = (horario.hora & 0x0F)+'0';    // separa o LSD da hora
      lcdc_escreve_char(digito);                      // escreve o dígito
      lcdc_escreve_char(':');
      digito = (horario.minuto>>4)+'0';               // separa o MSD do minuto
      lcdc_escreve_char(digito);                      // escreve o dígito
      digito = (horario.minuto & 0x0F)+'0';  // separa o LSD do minuto
      lcdc_escreve_char(digito);                      // escreve o dígito
      lcdc_escreve_char(':');
      digito = (horario.segundo>>4)+'0';     // separa o MSD do segundo
      lcdc_escreve_char(digito);                      // escreve o dígito
      digito = (horario.segundo & 0x0F)+'0'; // separa o LSD do segundo
      lcdc_escreve_char(digito);                      // escreve o dígito
   }
}
```

Exemplo 7-13

7.8. Técnicas de Baixo Consumo

Antes de encerrar a discussão sobre os microcontroladores MSP430, é preciso ressaltar alguns aspectos importantes para a operação de um microcontrolador com o menor consumo de corrente possível.

1. Utilize uma baixa freqüência de *clock*: quanto menor a freqüência de operação do microcontrolador menor o seu consumo de corrente. Tipicamente, um MSP430F449 consome cerca de 420μA/MHz quando operando a 3V. Este consumo varia quase linearmente com a variação da freqüência de *clock*. O fabricante especifica que a corrente em modo ativo pode ser determinada pela seguinte fórmula:

$$I_{ativo}=I_{1MHz}*f_{operação}$$

2. A tensão de operação também é um componente importante no consumo de corrente do microcontrolador: um MSP430F449 consome cerca de 420μA/MHz a 3V, mas quando operando a 2,2V, este consumo cai para aproximadamente 280μA/MHz. Novamente, cabe ao projetista determinar a menor tensão de operação possível para o circuito. Neste aspecto, é importante considerar as mínimas tensões de operação de alguns periféricos internos do microcontrolador (como os conversores A/D, a gravação da memória FLASH, etc.) e também dos eventuais periféricos externos. O consumo de corrente, como função da tensão de alimentação, pode ser determinado, aproximadamente, pela seguinte fórmula:

$$I_{ativo} = I_{3V} + 175\mu A/V * (V_{alimentação} - 3V)$$

3. Controle as fontes de *clock* do microcontrolador: os MSP430 possuem um excelente controle sobre o funcionamento do seu circuito de *clock*. Sempre que possível, utilize o ACLK como fonte de *clock* para os periféricos, uma vez que esse sinal permanece ativo em quase todos os modos de baixa potência (com exceção do LPM4), desta forma, será possível manter um ou mais periféricos funcionando, mesmo durante um modo de baixa potência.
4. Explore os recursos do DCO: sempre que possível, em vez de utilizar um oscilador de altafreqüência com cristal externo, procure utilizar o DCO. Ele possui características de baixo consumo, permitindo a alteração da freqüência de operação da CPU, de forma a otimizar o consumo de corrente.
5. Utilize ao máximo os periféricos internos: em muitos casos, é possível executar tarefas do *software* a partir dos periféricos internos, como no caso dos timers, que podem automatizar algumas tarefas de contagem e geração de sinais, o multiplicador por *hardware*, o DMA, etc. Utilizar um periférico para executar o trabalho da CPU significa poder colocá-la em um modo de baixa potência, mesmo que por breves instantes, o que contribui para a redução do consumo de energia.
6. Faça uso das interrupções para ativar a CPU em vez de mantê-la ativa todo o tempo: utilizando os vastos recursos de interrupção dos MSP430, é possível manter a CPU em um modo de baixa potência, sendo ativada somente na ocorrência de um evento de interrupção.
7. Desative os periféricos não utilizados: a arquitetura dos MSP430 permite desativar os periféricos que não estão sendo utilizados. Isso pode ser muito importante antes da entrada em um modo de baixa potência: se o conversor A/D é utilizado para converter uma tensão apenas quando o *chip* está ativo, é interessante desativá-lo antes da entrada no modo de baixa potência.

8. Utilize pinos de E/S do microcontrolador para controlar a alimentação de circuitos externos, desta forma é possível desativá-los antes da entrada em um modo de baixa potência.
9. Configure os pinos de E/S não utilizados para o modo de saída, ou conecte-os ao terra ou ao Vcc. Pinos configurados como entrada e deixados em aberto, além de servirem como uma excelente porta de entrada para ruídos, podem também apresentar correntes de fuga que contribuem para a elevação do consumo de corrente. Mantendo-os configurados como saída, ambos os efeitos são solucionados.

Estes são apenas alguns pontos a serem considerados para a redução do consumo de energia em uma aplicação microcontrolada. A seguir, veremos um exemplo de aplicação que utiliza princípios de baixa potência.

7.8.1. Relógio de Baixo Consumo

O exemplo seguinte utiliza o *display* LCD incluso com a placa Microlab X1 para implementar um relógio digital de 12 horas. Duas teclas do teclado da placa são utilizadas para o acerto das horas e minutos. Utilizamos a linha TCL_COL0 conectada diretamente ao terra, enquanto a linha TCL_LIN0 foi conectada ao pino P2.0 e a linha TCL_LIN1 foi conectada ao pino P2.1 (veja a figura 7-8 para maiores informações sobre o teclado). Neste caso, não se utiliza a varredura de teclado.

Foi utilizado um módulo de CPU MSP430F449 e graças ao emprego do modo LPM3 e dos periféricos de baixa potência do MSP, a aplicação toda em funcionamento normal consome cerca de 16μA/H.

O acesso ao controlador de LCD interno é feito com uma biblioteca escrita por Dado Sutter e gentilmente cedida para a inclusão neste livro. O arquivo do exemplo 7-14 deve ser salvo com o nome "lcds.h", enquanto o arquivo do exemplo 7-15 deve ser salvo como "lcds.c" na mesma pasta do projeto.

```
//*****************************************************************************
// Biblioteca de interface para o controlador de LCD dos MSP430
//*****************************************************************************
// Autor: Dado Sutter
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este biblioteca é uma implementação alternativa para a interface com o
// controlador de LCDs dos microcontroladores MSP430, operando no modo
// estático. As definições utilizadas são válidas para o display LCD incluso
// na placa Microlab X1
//*****************************************************************************

// Inicializa o controlador de LCDs
void lcds_ini(void);
// Escreve no primeiro dígito
void lcds_atualiza_digito_1(const unsigned char n);
// Escreve no segundo dígito
void lcds_atualiza_digito_2(const unsigned char n);
// Escreve no terceiro dígito
void lcds_atualiza_digito_3(const unsigned char n);
// Escreve no quarto dígito
void lcds_atualiza_digito_4(const unsigned char n);
// Apaga o display
void lcds_apaga_display(void);
```

Exemplo 7-14

```
//*****************************************************************************
// Biblioteca de interface para o controlador de LCD dos MSP430
//*****************************************************************************
// Autor: Dado Sutter
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este biblioteca é uma implementação alternativa para a interface com o
// controlador de LCDs dos microcontroladores MSP430, operando no modo está-
// tico. As definições utilizadas são válidas para o display LCD incluso na
// placa Microlab X1
//*****************************************************************************
#include "lcds.h"
#define LCDMEM ((char*) (0x0091))
char *LCD = LCDMEM; // acesso à memória do LCD como uma matriz C
// Cada linha corresponde aos valores que serão movidos para as posições
// de memória do controlador de LCD relativas à posição do dígito em
// questão. Cada rotina de atualização de um determinado dígito sabe onde
// colocar os seus dados.
// Pode facilmente ser otimizada pra uma tabela só e pra uma só função de
// atualização dos dígitos, se for necessário. Também o acesso às posições
// via ponteiros permitiria que vários bytes da tabela fossem atualizados de
// uma só vez na memória do LCD mas o sistema perderia muito em legibilidade.

// Decodificação do primeiro dígito (unidades)
const char lcds_1[]=
{
   0x11, 0x11, 0x10, 0x01, 0x00,     // 0
   0x11, 0x00, 0x00, 0x00, 0x00,     // 1
   0x01, 0x11, 0x10, 0x00, 0x10,     // 2
   0x11, 0x01, 0x10, 0x00, 0x10,     // 3
   0x11, 0x00, 0x00, 0x01, 0x10,     // 4
   0x10, 0x01, 0x10, 0x01, 0x10,     // 5
   0x10, 0x11, 0x10, 0x01, 0x10,     // 6
   0x11, 0x00, 0x10, 0x00, 0x00,     // 7
   0x11, 0x11, 0x10, 0x01, 0x10,     // 8
   0x11, 0x00, 0x10, 0x01, 0x10      // 9
};
// Decodificação do segundo dígito (dezenas)
const char lcds_2[]=
{
   0x11, 0x10, 0x11, 0x00, 0x01,     // 0
   0x01, 0x10, 0x00, 0x00, 0x00,     // 1
   0x10, 0x10, 0x01, 0x10, 0x01,     // 2
   0x11, 0x10, 0x01, 0x10, 0x00,     // 3
   0x01, 0x10, 0x10, 0x10, 0x00,     // 4
   0x11, 0x00, 0x11, 0x10, 0x00,     // 5
   0x11, 0x00, 0x11, 0x10, 0x01,     // 6
   0x01, 0x10, 0x01, 0x00, 0x00,     // 7
   0x11, 0x10, 0x11, 0x10, 0x01,     // 8
   0x01, 0x10, 0x11, 0x10, 0x00      // 9
};
// Decodificação do terceiro dígito (centenas)
const char lcds_3[]=
{
   0x01, 0x11, 0x10, 0x10, 0x00, 0x01,       // 0
   0x01, 0x01, 0x00, 0x00, 0x00, 0x00,       // 1
   0x00, 0x11, 0x00, 0x10, 0x10, 0x01,       // 2
   0x01, 0x11, 0x00, 0x00, 0x10, 0x01,       // 3
   0x01, 0x01, 0x10, 0x00, 0x10, 0x00,       // 4
   0x01, 0x10, 0x10, 0x00, 0x10, 0x01,       // 5
   0x01, 0x10, 0x10, 0x10, 0x10, 0x01,       // 6
   0x01, 0x11, 0x00, 0x00, 0x00, 0x00,       // 7
   0x01, 0x11, 0x10, 0x10, 0x10, 0x01,       // 8
   0x01, 0x11, 0x10, 0x00, 0x10, 0x00        // 9
};
// Decodificação do terceiro dígito (milhares)
const char lcds_4[]=
{
   0x10     // 1
};
// Inicialização do controlador e do timer básico
void lcds_ini (void)
{
  LCDCTL = LCDSTATIC + LCDP0 + LCDP1 + LCDP2 + LCDON;
  BTCTL = BT_fCLK2_ACLK;
}
```

```
// Esta versão só funciona para o layer 0, LCDs estáticos....
void lcds_atualiza_digito_1(const unsigned char n)
{
  LCD[0] = lcds_1[n*5];
  LCD[1] = lcds_1[n*5 + 1];
  LCD[2] = lcds_1[n*5 + 2];
  LCD[3] = lcds_1[n*5 + 3];
  LCD[4] = lcds_1[n*5 + 4];
}
void lcds_atualiza_digito_2(const unsigned char n)
{
  LCD[5] = lcds_2[n*5];
  LCD[6] = lcds_2[n*5 + 1];
  LCD[7] = lcds_2[n*5 + 2];
  LCD[8] = lcds_2[n*5 + 3];
  LCD[9] = lcds_2[n*5 + 4];
}
void lcds_atualiza_digito_3(const unsigned char n)
{
  LCD[10] = lcds_3[n*6];
  LCD[11] = lcds_3[n*6 + 1];
  LCD[12] = lcds_3[n*6 + 2];
  LCD[13] = lcds_3[n*6 + 3];
  LCD[14] = lcds_3[n*6 + 4];
  if (lcds_3[n*6 + 5])
    LCD[17] |= 0x01;  /* LCD100Data[n*6 + 5] */
  else
    LCD[17] &= 0xFE;  /* ~LCD100Data[n*6 + 5] */
}
void lcds_atualiza_digito_4(const unsigned char n)
{
  if (n) LCD[17] |= 0x10; else LCD[17] &= 0xEF;
}
void lcds_apaga_display(void)
{
  unsigned char temp;
  for (temp=0; temp<20; temp++)      LCD[temp] = 0;
}
```

Exemplo 7-15

```
//*****************************************************************************
// Relógio Digital Simples
//*****************************************************************************
// Autor: Fábio Pereira
// Para o livro Microcontroladores MSP430: Teoria e Prática
//*****************************************************************************
// Este programa demonstra a construção de um relógio digital simples
// utilizando um MSP430F449. O microcontrolador é mantido no modo
// LPM3 durante a maior parte do tempo, sendo ativado a cada 0,25s
// pelo temporizador básico (BT). Na função de tratamento de inter-
// rupção do mesmo, é medida a passagem de um segundo, contagem dos
// segundos, minutos e horas, além da monitoração das teclas.
//*****************************************************************************
#include <io430x44x.h>
#include <intrinsics.h>
#include "lcds.h"
#include "lcds.c"
unsigned char hora,minuto,segundo;
#define BTIP0 1
#define BTIP1 2
#define BTIP2 4
#define tcl_inc_hora P2IN_bit.P2IN_0
#define tcl_inc_minuto P2IN_bit.P2IN_1
//*****************************************************************************
// Função: trata_BT
//*****************************************************************************
// Esta função trata a interrupção do contador 2 do timer básico. O
// mesmo está configurado para dividir o ACLK por 256 * 32 = 8192,
// resultando em 4 interrupções por segundo (quando utilizando um
// cristal de 32768Hz no LFXT1). Nesta função também são monitoradas
// as teclas de incremento das horas e dos minutos
//*****************************************************************************
#pragma vector = BASICTIMER_VECTOR
__interrupt void trata_BT(void)
```

```
{
   static unsigned char contador, contador2, pontos, tempo_tecla;
   if (!tcl_inc_hora)
   {
   // se a tecla de incremento de horas estiver pressionada
      tempo_tecla++;         // incrementa o contador do tempo da tecla
      // se o contador for igual a 1 ou maior que 2
      if (tempo_tecla==1 || tempo_tecla>2)
      {
         hora++;    // incrementa a hora
         // ajusta o horário no formato BCD
         if ((hora & 0x0F)>0x09) hora=(hora & 0xF0)+0x10;
         // se a hora for maior que 11, volta a 0
         if (hora>0x11) hora = 0;
      }
   } else if (!tcl_inc_minuto)
   {
   // se a tecla de incremento de horas estiver pressionada
      tempo_tecla++;         // incrementa o contador do tempo da tecla
      // se o contador for igual a 1 ou maior que 2
      if (tempo_tecla==1 || tempo_tecla>2)
      {
         minuto++; // incrementa o minuto
         // ajusta o horário no formato BCD
         if ((minuto & 0x0F)>0x09) minuto=(minuto & 0xF0)+0x10;
         // se minuto for maior que 59, volta a 0
         if (minuto>0x59) minuto = 0;
      }
   // se nenhuma tecla pressionada, volta o contador de tempo das
   // teclas a zero
   } else tempo_tecla = 0;
   if (++contador>=4)
   {
      // se o contador de interrupções for maior ou igual a 4
      // passou-se um segundo
      contador = 0; // zera o contador
      segundo++;    // incrementa os segundos
      // ajusta a variável no formato BCD
      if ((segundo & 0x0F)>0x09) segundo=(segundo & 0xF0)+0x10;
      if (segundo>0x59)      // se segundos maior que 59
      {
         segundo = 0;        // zera segundos
         minuto++; // incrementa minutos
         // ajusta a variável no formato BCD
         if ((minuto & 0x0F)>0x09) minuto=(minuto & 0xF0)+0x10;
         if (minuto>0x59)   // se minutos maior que 59
         {
            minuto = 0;     // zera minutos
            hora++;         // incrementa hora
            // ajusta a variável no formato BCD
            if ((hora & 0x0F)>0x09) hora=(hora & 0xF0)+0x10;
            // se hora maior que 11, zera a hora
            if (hora>0x11) hora = 0;
         }
      }
      // apresenta o MSD da hora
      lcds_atualiza_digito_4(hora>>4);
      // apresenta o LSD da hora
      lcds_atualiza_digito_3(hora & 0x0F);
      // apresenta o MSD do minuto
      lcds_atualiza_digito_2(minuto>>4);
      // apresenta o LSD do minuto
      lcds_atualiza_digito_1(minuto & 0x0F);
   }
   if (++contador2>=2)
   {
      // se passou meio segundo
      contador2 = 0;
      pontos = !pontos;      // inverte o estado dos pontos
      // se os pontos estiverem ativos, acende-os no display
      if (pontos) LCDM9^=0x01;
   }
   if (tempo_tecla)
   {
      // se tempo_tecla maior que zero, alguma tecla está
      // pressionada, neste caso, devemos atualizar os
```

```
        // dígitos do display
        lcds_atualiza_digito_4(hora>>4);
        lcds_atualiza_digito_3(hora & 0x0F);
        lcds_atualiza_digito_2(minuto>>4);
        lcds_atualiza_digito_1(minuto & 0x0F);
    }
}
void main(void)
{
    unsigned int tempo1, tempo2;
    WDTCTL = WDTPW+WDTHOLD;         // desliga o watchdog
    // configura o FLL para aproximadamente 1MHz
    // (a partir do LFXT1 a 32768Hz)
    SCFI0 = FLLD_1;
    SCFQCTL = 0x1E;
    FLL_CTL0 = DCOPLUS;
    FLL_CTL1 = SMCLKOFF + XT2OFF;
    while (!IFG1_bit.OFIFG) IFG1_bit.OFIFG = 0;
    // configura o timer básico
    BTCTL = BTDIV + BT_fCLK2_DIV32;   // ACLK / (256 * 32)
    IE2 = BTIE;       // ativa a interrupção do timer básico
    // liga o LCD, seleciona o modo estático
    // habilita todas as saídas do módulo LCD
    LCDCTL = LCDON+LCDSTATIC+LCDP2+LCDP1+LCDP0;
    // configura todos os pinos como saída, para reduzir o consumo
    P1DIR = P3DIR = P4DIR = P5DIR = P6DIR = 0xFF;
    P2DIR = 0xFC;
    // inicia em zero a contagem
    hora = 0x00;
    minuto = 0x00;
    segundo = 0x00;
    __enable_interrupt();     // habilita as interrupções
    while (1)
    {
            __low_power_mode_3();     // entrada no modo LMP3
    }
}
```

Exemplo 7-16



Conjunto de Instruções *Assembly*

Mnemônico		Descrição	Operação	*Flags*			
				V	N	Z	C
ADC(.B)	dest	Adiciona C ao destino	dest = dest + C	*	*	*	*
ADD(.B)	fonte,dest	Adiciona a fonte ao destino	dest = fonte + dest	*	*	*	*
ADDC(.B)	fonte,dest	Adiciona a fonte mais o C ao destino	dest = fonte + C + dest	*	*	*	*
AND(.B)	fonte,dest	Operação lógica AND da fonte com o destino	dest = fonte AND dest	0	*	*	*
BIC(.B)	fonte,dest	Apaga os *bits* da fonte no destino	dest = not(fonte) AND dest	-	-	-	-
BIS(.B)	fonte,dest	Seta os *bits* da fonte no destino	dest = fonte OR dest	-	-	-	-
BIT(.B)	fonte,dest	Testa os *bits* da fonte no destino	dest AND fonte	0	-	-	-
BR	dest	Desvia para o destino	PC = dest	-	-	-	-
CALL	dest	Chamada de sub-rotina no destino	Pilha = PC + 2, PC = dest	-	-	-	-
CLR(.B)	dest	Apaga o conteúdo do destino	dest = 0	-	-	-	-
CLRC		Apaga o *flag* C	C = 0	-	-	-	0
CLRN		Apaga o *flag* N	N = 0	-	0	-	-
CLRZ		Apaga o *flag* Z	Z = 0	-	-	0	-
CMP(.B)	fonte,dest	Compara a fonte e o destino	dest - src	*	*	*	*
DADC(.B)	dest	Adição decimal do C com o destino	dest = dest + C (decimal)	*	*	*	*
DADD(.B)	fonte,dest	Adição decimal da fonte com o C ao destino	dest = fonte + C + dest (decimal)	*	*	*	*
DEC(.B)	dest	Decrementa o destino	dest = dest - 1	*	*	*	*
DECD(.B)	dest	Decremento duplo do destino	dest = dest - 2	*	*	*	*
DINT		Desabilita as interrupções	GIE = 0	-	-	-	-
EINT		Habilita as interrupções	GIE = 1	-	-	-	-
INC(.B)	dest	Incrementa o destino	dest = dest + 1	*	*	*	*
INCD(.B)	dest	Incremento duplo do destino	dest = dest + 2	*	*	*	*
INV(.B)	dest	Complementa os *bits* do destino	dest = NOT (dest)	*	*	*	*
JC/JHS	label	Pula se C = 1 / Pula se maior ou igual		-	-	-	-
JEQ/JZ	label	Pula se igual / Pula se Z = 1		-	-	-	-
JGE	label	Pula se maior ou igual		-	-	-	-
JL	label	Pula se menor		-	-	-	-
JMP	label	Pula	PC = PC + 2 * *offset*	-	-	-	-
JN	label	Pula se negativo (se N = 1)		-	-	-	-
JNC/JLO	label	Pula se C = 0 / Pula se menor		-	-	-	-
JNE/JNZ	label	Pula se diferente / Pula se Z = 0		-	-	-	-
MOV(.B)	fonte,dest	Copia o conteúdo de fonte para destino	dest = fonte	-	-	-	-
NOP		Nenhuma operação		-	-	-	-
POP(.B)	dest	Lê um dado da pilha e guarda no destino	dest = @SP, SP = SP + 2	-	-	-	-
PUSH(.B)	fonte	Armazena o conteúdo da fonte na pilha	SP = SP – 2, @SP = fonte	-	-	-	-

Mnemônico		Descrição	Operação	*Flags*			
				V	N	Z	C
RET		Retorna de uma sub-rotina	PC = @SP, SP = SP + 2	-	-	-	-
RETI		Retorna de uma interrupção		*	*	*	*
RLA(.B)	dest	Rotação aritmética do destino à esquerda		*	*	*	*
RLC(.B)	dest	Rotação do destino à esquerda através do *Carry*		*	*	*	*
RRA(.B)	dest	Rotação aritmética do destino à direita		0	*	*	*
RRC(.B)	dest	Rotação do destino à direita através do *Carry*		*	*	*	*
SBC(.B)	dest	Subtrai o C invertido do destino	dest = dest + 0xFFFF + C	*	*	*	*
SETC		Seta o *flag* C	C = 1	-	-	-	1
SETN		Seta o *flag* N	N = 1	-	1	-	-
SETZ		Seta o *flag* Z	Z = 1	-	-	1	-
SUB(.B)	fonte,dest	Subtrai a fonte do destino	dest = dest – fonte	*	*	*	*
SUBC(.B)	fonte,dest	Subtrai a fonte e o C invertido do destino	dest = dest – fonte – NOT(C)	*	*	*	*
SWPB	dest	Troca os *bytes* do destino		-	-	-	-
SXT	dest	Extensão do sinal do destino		0	*	*	*
TST(.B)	dest	Testa o destino	dest = 0 ?	0	*	*	1
XOR(.B)	fonte,dest	Operação lógica XOR da fonte com o destino	dest = fonte XOR dest	*	*	*	*

B

Funções da Biblioteca C

Em seguida apresentamos algumas funções da biblioteca-padrão da linguagem C, disponíveis no compilador IAR.

Nome	Retorna	Biblioteca	Descrição
abort (void)	void	stdlib.h	Termina o programa. Todas as funções registradas com atexit são executadas e em seguida é executada um HALT
abs (int x)	int	stdlib.h	Retorna o valor absoluto de x
acos (double x)	double	math.h	Calcula o arco cosseno de x
acosf (float x)	float	math.h	Calcula o arco cosseno de x
asin (double x)	double	math.h	Calcula o arco seno de x
asinf (float x)	float	math.h	Calcula o arco seno de x
assert (int cond)	void	assert.h	Emite mensagem caso a condição (cond) seja verdadeira
atan (double x)	double	math.h	Calcula o arco tangente de x
atanf (float x)	float	math.h	Calcula o arco tangente de x
atan2 (double x , double y)	double	math.h	Calcula o arco tangente de y / x
atan2f (float x , float y)	float	math.h	Calcula o arco tangente de y / x
atexit (void (*func) (void))	int	stdlib.h	Permite instalar uma função que será chamada antes do encerramento do programa. Até 32 funções podem ser registradas com esta função. A ordem de chamada delas no encerramento é inversa à ordem de registro. A função atexit irá retornar 0 caso tenha conseguido registrar a função; caso negativo, será retornado um valor diferente de zero.
atof (const char *s)	double	stdlib.h	Converte a string *s em um número double. A função aceita os seguintes caracteres: 0,1,2,3,4,5,6,7,8,9,–,+,e,E.
atoi (const char *s)	int	stdlib.h	Converte a string *s em um inteiro
atol (const char *s)	long int	stdlib.h	Converte a string *s em um inteiro longo
bsearch (const void *key, const void *array, size_t n, size_t size, cmp_func cmp)	void *	stdlib.h	Realiza uma busca binária em uma matriz ordenada. Ela chama a função de comparação cmp com dois argumentos: um ponteiro para o elemento chave a ser encontrado e um ponteiro para o elemento da matriz.
calloc (size_t n, size_t size)	void *	stdlib.h	Aloca um bloco de memória contendo n elementos de tamanho size. Todos os elementos são inicializados com zero. Para liberar a memória alocada, utilize free(). Esta função não deve ser utilizada em ISRs
ceil (double x)	double	math.h	Retorna o menor inteiro acima de x
ceilf (float x)	float	math.h	Retorna o menor inteiro acima de x
cos (double x)	double	math.h	Calcula o cosseno de x radianos
cosf (float x)	float	math.h	Calcula o cosseno de x radianos
cosh (double x)	double	math.h	Calcula o cosseno hiperbólico de x radianos
coshf (float x)	float	math.h	Calcula o cosseno hiperbólico de x radianos
div (int x, int y)	div_t	stdlib.h	Calcula tanto o quociente como o resto da divisão de x por y
exit (int status)	void	stdlib.h	Termina o programa normalmente. Todas as funções registradas com atexit são executadas e em seguida é executada um HALT. O argumento status é ignorado
exp (double x)	double	math.h	Calcula e^x
expf (float x)	float	math.h	Calcula e^x

Nome	Retorna	Biblioteca	Descrição
fabs (double x)	double	math.h	Retorna o valor absoluto de x
fabsf (float x)	float	math.h	Retorna o valor absoluto de x
floor (double x)	double	math.h	Retorna o menor valor inteiro abaixo de x
floorf (float x)	float	math.h	Retorna o menor valor inteiro abaixo de x
fmod (double x, double y)	double	math.h	Retorna o resto da divisão float de x por y
fmodf (float x, float y)	float	math.h	Retorna o resto da divisão float de x por y
free (void *ptr)	void	stdlib.h	Libera a memória alocada pela função calloc, malloc ou realloc. Esta função não deve ser utilizada em ISRs
frexp (double x, int *y)	double	math.h	Retorna o componente exponencial e fracionário de x
frexpf (float x, int *y)	float	math.h	Retorna o componente exponencial e fracionário de x
getchar (void)	int	stdio.h	Retorna um caractere lido do dispositivo padrão de entrada stdin
gets (char *s)	char *	stdio.h	Retorna um ponteiro para uma string de caracteres "s", lida do dispositivo padrão de entrada stdin
isalnum (int carac)	int	ctype.h	Testa se o argumento é uma letra ou número
isalpha (int carac)	int	ctype.h	Testa se o argumento é uma letra
isdigit (int carac)	int	ctype.h	Testa se o argumento é um número
islower (int carac)	int	ctype.h	Testa se o argumento é uma letra minúscula
isspace (int carac)	int	ctype.h	Testa se o argumento é um espaço
isupper (int carac)	int	ctype.h	Testa se o argumento é uma letra maiúscula
isxdigit (int carac)	int	ctype.h	Testa se o argumento é um dígito hexadecimal
iscntrl (int carac)	int	ctype.h	Testa se o argumento é um caractere de controle
isgraph (int carac)	int	ctype.h	Testa se o argumento é um caractere gráfico
isprint (int carac)	int	ctype.h	Testa se o argumento é imprimível
ispunct (int carac)	int	ctype.h	Testa se o argumento é um caractere de pontuação
labs (long x)	long	stdlib.h	Retorna o valor absoluto de x
ldexp (double x , int y)	double	math.h	Retorna $x*2^y$
ldexpf (float x , int y)	float	math.h	Retorna $x*2^y$
ldiv (long int x, long int y)	ldiv_t	stdlib.h	Calcula tanto o quociente como o resto da divisão de x por y
log (double x)	double	math.h	Calcula o logaritmo natural de x
logf (float x)	float	math.h	Calcula o logaritmo natural de x
log10 (double x)	double	math.h	Calcula o logaritmo base 10 de x
log10f (float x)	float	math.h	Calcula o logaritmo base 10 de x
longjmp (jmp_buf env, int val)	void	setjmp.h	Desvia o programa para um outro local previamente definido com setjmp.
malloc (size_t size)	void *	stdlib.h	Aloca um bloco de memória para um objeto de tamanho igual a size *bytes*. O conteúdo da memória é indefinido. Para liberar a memória, utilize free(). Esta função não deve ser utilizada em ISRs.
memchr (const void *p, int ch, size_t n)	void *	string.h	Procura a primeira ocorrência de um *byte* contendo (ch & 0xFF) nos primeiros n *bytes* da área de memória apontada por p. Retorna um ponteiro para a primeira ocorrência ou nulo para nenhuma ocorrência.
memcmp (const void *p, const void *q, size_t n)	void *	string.h	Compara os primeiros n *bytes* das área de memória apontadas por p e q. Retorna um valor positivo se p > q, negativo se p < q e zero se p = q
memcpy (const void *p, const void *q, size_t n)	void *	string.h	Copia n *bytes* de q para p
memmcpy (const void *p, const void *q, size_t n)	void *	string.h	Copia n *bytes* de q para p
memset (void *p, int val, size_t n)	void *	string.h	Seta n *bytes* do ponteiro (p) com o valor (val & 0xFF). Retorna p
modf (doublet x, double *y)	double	math.h	Retorna a parte inteira (retornada por *y) e fracionária de x
modff (float x, float *y)	float	math.h	Retorna a parte inteira (retornada por *y) e fracionária de x
perror (const char *msg)	void	stdio.h	Escreve uma mensagem de erro no dispositivo de erro padrão
pow (double x, double y)	double	math.h	Retorna x^y
powf (float x, float y)	float	math.h	Retorna x^y
printf (const char *format, ...)	int	stdio.h	Escreve um texto formatado de acordo com a string "format" no dispositivo de saída padrão stdout
putchar (int c)	int	stdio.h	Escreve o caractere "c" no dispostivo de saída padrão stdout

Nome	Retorna	Biblioteca	Descrição
puts (const char *s)	int	stdio.h	Escreve os caracteres da string "s" no dispositivo padrão de saída stdout
rand (void)	int	stdlib.h	Gera um número pseudo-randômico entre o e RAND_MAX. A "semente" pode ser alterada utilizando a função srand()
realloc (void *ptr, size_t size)	void *	stdlib.h	Modifica o tamanho do bloco de memória, preservando o seu conteúdo. ptr precisa ser um ponteiro retornado por calloc(), malloc() ou realloc(). Caso o ponteiro seja nulo, ela se comporta como malloc(). Se o novo tamanho for menor que o anterior, a parte excedente ao final do bloco é descartada. Ela retorna nulo caso não exista memória suficiente, ou um ponteiro para o novo bloco de memória.
setjmp (jmp_buf env)	int	setjmp.h	Salva o estado atual do programa no *buffer* env e retorna zero. Esse *buffer* pode ser posteriomente utilizado com longjmp() para retornar o programa ao estado e ponto em que se encontrava.
sin (double x)	double	math.h	Retorna o seno de um ângulo de x radianos
sinf (float x)	float	math.h	Retorna o seno de um ângulo de x radianos
sinh (double x)	double	math.h	Retorna o seno hiperbólico de x
sinhf (float x)	float	math.h	Retorna o seno hiperbólico de x
sprintf (char *s, const char *format, ...)	int	stdio.h	Escreve uma string formatada e terminada por nulo em s.
sqrt (double x)	double	math.h	Retorna a raiz quadrada de x
sqrtf (float x)	float	math.h	Retorna a raiz quadrada de x
srand (unsigned int seed)	void	stdlib.h	Inicializa a semente do gerador de números randômicos com o valor seed
sscanf (const char *s, const char *format, ...)	int	stdio.h	Lê a string s e retorna os seus parâmetros de formato
strcat (char *p, const char *q)	char *	string.h	Concatena a string q em p
strchr (const char *p, int ch)	char *	string.h	Retorna um ponteiro para a ocorrência do caractere ch na string p
strcmp (const char *p, const char *q)	int	string.h	Compara a string p com a string q e retorna um valor negativo caso p menor que q, zero caso p = q e um valor positivo caso p maior que q
strcpy (char *p, const char *q)	char *	string.h	Copia a string q para a string p
strcspn (const char *p, const char *q)	size_t	string.h	Procura na string p pelo primeiro caractere que também aparece em q
strerror (int errno)	char *	string.h	Retorna uma mensagem de erro equivalente ao código de erro em errno
strlen (const char *s)	size_t	string.h	Conta o número de caracteres em s
strncat (char *p, const char *q, size_t n)	char *	string.h	Concatena a string q na string p. Se q contém mais de n caracteres, somente os primeiros n caracteres de q são concatenados à string p
strncmp (char *p, const char *q, size_t n)	char *	string.h	Compara os n primeiros caracteres da string p com a string q
strncpy (char *p, const char *q, size_t n)	char *	string.h	Copia os primeiros n caracteres da string q para a string p, sobrescrevendo o conteúdo prévio de p. Caso q tenha menos de n caracteres, são adicionados nulos
strrchr (const char *s, int c)	char *	string.h	Retorna um ponteiro para a ocorrência do caractere c na string s, inicia a procura pelo final da string
strspn (const char *p, const char *q)	size_t	string.h	Retorna o tamanho da parte inicial de p que contém somente caracteres que também aparecem em q. A função retorna a posição do primeiro caractere de p que não aparece em q.
strstr (const char *p, const char *q)	char *	string.h	Procura a string q na string p. Retorna um ponteiro para o início da primeira ocorrência de q em p
strtod (const char *s, char **end)	double	stdlib.h	Converte a string s em um número de ponto flutuante double
strtok (char *p, const char *q)	char *	string.h	Procura a próxima ocorrência de um sinal definido em s2 na string s1
strtol (const char *s, char **end, int base)	long	stdlib.h	Converte a string (s) em um número inteiro longo na base (base)
strtoul (const char *s, char **end, int base)	unsigned long	stdlib.h	Converte a string s em um número inteiro longo sem sinal
tan (double x)	double	math.h	Calcula a tangente de x radianos
tanf (float x)	float	math.h	Calcula a tangente de x radianos
tanh (double x)	double	math.h	Calcula a tangente hiperbólica de x radianos
tanhf (float x)	float	math.h	Calcula a tangente hiperbólica de x radianos
tolower (int ch)	int	ctype.h	Retorna o caractere ch em minúsculo
toupper (int ch)	int	ctype.h	Converte o caractere ch em maiúsculo
va_arg, va_end, va_start	void	stdarg.h	Macros para recebimento de um número variável de lista de parâmetros de funções.

É possível especificar funções para implementar os dispositivos de entrada, saída e erro padrões, utilizando a seguinte sintaxe:

```
#define stdin <ponteiro para a função>
#define stdout <ponteiro para a função>
#define stderr <ponteiro para a função>
```

Além disso, dependendo da biblioteca utilizada (CLIB ou DLIB) e das opções dela, as funções de entrada/saída formatada, tais como: printf, scanf, etc., implementam mais ou menos códigos de formatação. Na tabela seguinte, temos os códigos de formatação suportados.

Código	Formato de saída	Código	Formato de saída
%c	unsigned char	**%n**	ponteiro para int
%d	int	**%hn**	ponteiro para short
%hd	short int	**%ln**	ponteiro para long
%ld	long int	**%o**	int em formato octal
%e	double	**%ho**	short int em formato octal
%Le	long double	**%lo**	long int em formato octal
%E	double	**%p**	ponteiro para nulo
%LE	long double	**%s**	string
%f	double	**%u**	unsigned int
%Lf	long double	**%hu**	unsigned short int
%g	double	**%lu**	unsigned long int
%Lg	long double	**%x**	unsigned int em formato hexadecimal
%G	double	**%hx**	unsigned short int em formato hexadecimal
%LG	long double	**%lx**	unsigned long int em formato hexadecimal
%i	int	**%X**	unsigned int em formato hexadecimal
%hi	short int	**%hX**	unsigned short int em formato hexadecimal
%li	long int	**%lX**	unsigned long int em formato hexadecimal
%%	imprime o símbolo "%"		

DAC12_xCTL 269
DAC12_xDAT 271
DADC 81
DADD 80
DCO 116, 118, 352
DCOCTL 127
DEBUG 46
Debugger 39
DEC 85
DECD 86
depuração 39, 322
DINT 102
Disassembly 43
Display LCD Gráfico 367
DLIB 348, 407
DMA 287
DMACTL0 294
DMACTL1 295
DMAxCTL 295
DMAxDA 297
DMAxSA 297
DMAxSZ 297
double 325, 338, 350

E

Embedded Workbench 35
enum 329

F

FCTL1 306
FCTL2 307
FCTL3 308
FET 39, 317
FLL 116, 120, 135
FLL_CTL0 131
FLL_CTL1 132
float 325, 338, 350

G

GIE 26, 102
GPR 27

H

Heap 51
HT1381 ..... 197, 390

I

I²C 199
  byte de partida 202
  chamada geral 201
  formato dos dados 200
  modo de endereçamento de 10 bits 202
  protocolo 199
I2CDCTL 212
I2CDRB 214
I2CDRW 214
I2CIE 215
I2CIFG 216
I2CIV 216
I2CNDAT 215
I2COA 214
I2CPSC 213
I2CSA 215
I2CSCLH 213
I2CSCLL 213
I2CTCTL 211
IE1 ..... 111, 133, 185, 196, 309, 314, 316
IE2 ..... 111, 169, 185, 196
IEE754 338
IFG1 ..... 112, 133, 186, 197, 314, 316
IFG2 ..... 112, 169, 186, 197
INC 84
INCD 84
inline 347
Instruções
  desvio 55
  dois operandos 54
  emuladas 68
  modo absoluto 63
  modo imediato 57
  modo indexado 61
  modo indireto 64
  modo indireto com auto-incremento 66
  modo registrador 59
  modo simbólico 62
  modos de endereçamento 56
  temporização 103
  um operando 53
int 325
Interrupções 107
  categorias 107
  desabilitação 102, 115
  habilitação 102, 115
  latência 107
  não mascaráveis 108
  registradores 111
  tratamento 112
  vetores 108
  vetores em C 115
intrinsics 115
INV 90
ISR 107

J

JC 96
JEQ 96
JGE 97
JHS 96
JL 98
JLO 97
JMP 95
JN 98
JNC 97
JNE 96
JNZ 96
JTAG 22, 316
JZ 96

L

LCD 117, 166, 272, 397
LCDCTL 278
LCDMx 279
Live Watch 43, 49
Locals 43
Long 325
Long long 338
LPM0 ..... 29, 126
LPM1 ..... 29, 126
LPM2 29
LPM3 ..... 29, 397
LPM4 29

M

MAC 283, 284
MACS 284
MAX232 ..... 21, 384
MCLK 116, 122, 126, 142
ME1 ..... 184, 196
ME2 ..... 185, 196
Memória
  Organização 27
  ROM 28
  visualização do conteúdo 43
Memória FLASH
  apagamento 302
  memória de informação 303
  programação 304
  programação em blocos 305
  segmentos 302
Modificadores
  auto 326
  const 326
  extern 326
  register 326
  static 326
  volatile 326
Modos de baixo consumo 28
Modulador 119, 121, 172
Módulo LCD caractere
  CGRAM 356
  comandos 358
  DDRAM 356
  pinagem 356
  tabela de caracteres 357
MOV 47, 66, 68, 72
MPY 284
MPYS 284
MSP430
  CPU 23
  família 1xx 30
  família 2xx 30
  família 3xx 30
  família 4xx 30
  família 5xx 30
  famílias 30
  instruções assembly 53
  modos de operação 28
  organização da memória 27
MSP430F149 18
MSP430F449 19
MSPGCC 35
Multiplicador por Hardware 283

N

N 26
NMI 108, 315
NMIIE 108
NOP 102

O

OAxCTL0 236
OAxCTL1 237
OFIE 108
OFIFG 108, 126, 127, 133
OP2 285
Op-codes 53
Oscilador 116
  DCO 118
  gerenciamento de falha 124
  LFXT1 116
  XT2 118
OSCOFF 26, 28
OUTPUT 38

P

P1DIR 140
P1IES 141
P1IFG 140
P1IN 140
P1OUT 140
P1REN 141
P1SEL 140
P2DIR 140
P2IES 141
P2IFG 140

P2IN 140
P2OUT 140
P2REN 141
P2SEL 140
P3DIR 140
P3IN 140
P3OUT 140
P3REN 141
P3SEL 140
P4DIR 140
P4IN 140
P4OUT 140
P4REN 141
P4SEL 140
P5DIR 140
P5IN 140
P5OUT 140
P5REN 141
P5SEL 140
P6DIR 140
P6IN 140
P6OUT 140
P6REN 141
P6SEL 140
pack(x) 339
pilha 24, 44, 73, 74, 98, 99, 106, 351
POP 24, 74
POR 105
Portas de E/S 138
  interrupção 138
printf 51, 52, 348, 407
Profiling 44
Projeto
  compilando um projeto em C 51
  montagem em assembly 41
  novo projeto em assembly 36
  novo projeto em C 50
  opções 38
  simulação 42
  Simulando um projeto em C 51
PUC 105
PUSH 24, 73
putchar 52
PWM 146, 156, 382

R

Registradores
  convenções adotadas 21
  da CPU 23
  filtros de visualização 45
  geradores de constantes 27, 68
  propósito geral 24, 27
  R1 24
  R2 25
  SP 24, 66
  SR 25, 28
  visualização 43
reset 105, 106, 315
RESHI 283, 285
RESLO 283, 285
RET 99
RETI 99
RLA 86
RLC 90
RRA 87
RRC 92
RTC 120, 166, 197, 390
RTI 28, 107
RUN TO CURSOR 48

S

SAR 251
SBB ..... Consulte SUBC
SBC 83
scanf 51, 348, 407
SCFGCTL 130
SCFI0 131
SCFI1 131
SCG0 ..... 26, 28
SCG1 ..... 26, 28
Servo 382
SETC 101
SETN 101
SETZ 101
SFR 44
short 325
signed 325
sizeof 327
SMCLK 116, 123, 126, 142
SPI 188, 390
STEP INTO 48
STEP OUT 48
SUB 81
SUBC 82
SUMEXT 283, 285
Supervisor de Tensão ..... Consulte SVS
SVS 300
  pinos de conexão 301
SVSCTL 301
SWPB 73
SXT 88

T

T6963C 367
  comandos 368
TACCRx 146, 153
TACCTLx 145, 152
TACTL 151
TAIFG 144
TAIV 149, 154
TAR 143, 152
TBCCRx 158, 165
TBCCTLx 163
TBCLx 159
TBCTL 162
TBIV 160, 165
TBR 158, 163
Teclado 380
Técnicas de Baixo Consumo 396
Temporizador Básico 166
  interrupção 167
Terminal de I/O 43, 52
Timer 1 166
Timer A 143, 355
  interrupções 149
  modo de captura 145
  modo de comparação/PWM 146
  modos de contagem 144
  pinos de conexão 149
  reset 145
Timer B 157
  agrupamento de canais 159
  interrupções 160
  largura programável 158
  latches de comparação 158
  pinos de conexão 160
  saídas em alta impedância 160
Trace 44
TST 94
typedef 333

U

U0CTL 210
ULA 77
unsigned 325
USART 384
USART assíncrona 171
  configuração 175
  endereçamento idle-line 177
  endereçamento por bit 179
  erros de recepção 176
  gerador de baud rate 172
  interrupções 180
  pinos de conexão 180
  recepção 176
  tabela de velocidades 173
  transmissão 175
USART I^2C 199
  arbitragem 207
  características do hardware 202
  configuração 204
  gerador de clock 204
  interrupções 209
  modo escravo 207
  modo mestre 205
  pinos de conexão 209
  utilização com DMA 208
USART SPI 188
  configuração 190
  gerador de clock 190
  interrupções 192
  modo escravo 191
  modo mestre 191
  modos 189
  pinos de conexão 192
UxBR0 ..... 183, 195
UxBR1 195
UxCTL 181, 193
UxMCTL 183, 195
UxRCTL 182
UxRXBUF 184, 195
UxTCTL 182, 194
UxTXBUF 184, 195

V

V 26
Void 324

W

Watch 43
Watchdog 106, 355, 311
WDT 311
WDT+ 312
WDTCTL 313, 315
Workspace 36

X

XOR 89

Z

Z 26

Referências Bibliográficas

1. PEREIRA, F. **Microcontroladores HC908Q:** *Teoria e Prática*. São Paulo: Érica, 2004.
2. _____. **Microcontroladores PIC:** *Programação em C*. São Paulo: Érica, 2003.
3. _____. **Microcontroladores PIC:** *Técnicas Avançadas*. São Paulo: Érica, 2002.
4. NAGY, C. **Embedded Systems Design Using the TI MSP430 Series**. USA: Newnes, 2003.
5. PREDKO, M. **Handbook of Microcontrollers**. USA: McGraw Hill, 1998.
6. MANZANO, J.A.N.G. **Estudo Dirigido de Linguagem C**. São Paulo: Érica. 2002.
7. RITCHIE, D.M.; KERNIGHAN, B.W. **The C Programming Language**. USA: Prentice Hall, 1989.
8. SCHILDT, H. **C Completo e Total**. São Paulo: McGraw Hill, 1990.

Manuais:

9. Texas Instruments. **MSP430x1xx User's Manual - SLAU049E**. USA: Texas Instruments, 2005.
10. _____. **MSP430x1xx Family User's Guide - Errata - SLAZ007**. USA: Texas Instruments, 2004.
11. _____. **MSP430x2xx User's Manual - SLAU144**. USA: Texas Instruments, 2004.
12. _____. **MSP430x4xx User's Manual - SLAU056D**. USA: Texas Instruments, 2004.
13. _____. **MSP430x4xx Family User's Guide - Errata - SLAZ008**. USA: Texas Instruments, 2004.
14. BIERL, L. **MSP430 Family Mixed-Signal Microcontroller Application Reports - SLAA024**. USA: Texas Instruments, 2000.
15. IAR. **ICC MSP430 C Compiler Programming Guide**. USA: Human-Computer Interface, 1996.
16. Texas Instruments. **MSP430 Family Serial Programming Adapter Manual - SLAU048D**. USA: Texas Instruments, 2003.
17. Diversos *datasheets* da Texas Instruments.

Notas de aplicação:

18. MUEHLHOFER, A. **Controlling the DCO Frequency of the MSP430x11x - SLAA074**. Alemanha: Texas Instruments, 2000.
19. BICCINI, M. **MSP430F21x1 Architecture Summary - SLAA217**. USA: Texas Instruments, 2004.
20. RAJU, M. **Digital FIR Filter Design Using the MSP430F16x - SLAA228**. USA: Texas Instruments, 2004.
21. MITCHELL, M. **Implementing a Real-Time Clock on the MSP430 - SLAA076A**. USA: Texas Instruments, 2001.
22. GRAF, F. **Features of the MSP430 Bootstrap Loader - SLAA089B**. USA: Texas Instruments, 2003.
23. RZEHAK, V. **Application of Bootstrap Loader in MSP430 With Flash Hardware and Software Proposal - SLAA096B**. USA: Texas Instruments, 2001.
24. RAJU, MURUGAVEL. **Solid State Voice Recorder Using Flash MSP430 - SLAA123**. USA: Texas Instruments, 2001.
25. BRENNER, N. e MUZZARELLI, C.. **Implementing An Ultralow-power Thermostat With Slope A/D Conversion - SLAA129A**. USA: Texas Instruments, 2004.
26. BIERL, L. **Interfacing the 3-V MSP430 to 5-V Circuits - SLAA148**. USA: Texas Instruments, 2002.
27. ALBUS, Z. & GRAF, F. & KOESLER, M. **Programming a Flash-Based MSP430 Using the JTAG Interface - SLAA149**. USA: Texas Instruments, 2002.
28. DANNENBERG, A. **MSP430 Isolated FET Interface**. USA: Texas Instruments, 2003.

Apresentações em PDF:

29. **430 DAY - 2004 - SLAC037A**, Texas Instruments.
30. **MSP430 Design Seminar**, Texas Instruments.
31. **Meet the MSP430: An Introduction to the MSP430 Ultra-Low-Power MCU**, Texas Instruments.

Sites na Internet:

32. http://www.ti.com - Site da Texas Instruments, fabricante do MSP430 e do ambiente *Code Composer Essentials.*
33. http://www.iar.com - Site do fabricante da ferramenta de programação (Embedded Workbench).
34. http://www.quadravox.com/AQ430.htm - Ambiente de programação.
35. http://www.rowley.co.uk - Ambiente de programação Crossworks.
36. http://www.imagecraft.com - Ambiente de programação.
37. http://mspgcc.sourceforge.net - Ambiente de programação GNU (MSPGCC).
38. http://www.asm51.eng.br/forum/default.asp - Fórum (web) sobre diversos microcontroladores, incluindo o MSP430.
39. http://www.eletronica.etc.br/piclistbr/index.php - Lista de discussão por e-mail, sobre microcontroladores e eletrônica.
40. http://reniemarquet.sites.uol.com.br/ - site com alguns tutoriais muito interessantes, incluindo um sobre módulos LCD.
41. http://pdf.toshiba.com/taec/components/Datasheet/T6963CDS.pdf - datasheet do controlador T6963C.
42. http://www.densitron.com/editor/pdfs/t6963appnotes1995.pdf - uma excelente nota de aplicação descrevendo a programação do T6963.
43. http://www.fatlion.com/sailplanes/servos.html - pinagem de servos R/C.
44. http://www.seattlerobotics.org/encoder/200009/Servos.html - circuitos de servos R/C.
45. http://www.fatlion.com/sailplanes/servochart.html - planilha comparativa entre servos R/C de diversos fabricantes.
46. http://hp.vector.co.jp/authors/VA002416/teraterm.html - programa Tera Term para comunicação serial.
47. http://www.rowley.co.uk/msp430/basic.htm - um interpretador BASIC para o MSP430.

Marcas Registradas

MSP430 é uma marca registrada da Texas Instruments, Inc.

Embedded Workbench é uma marca registrada da IAR Systems.

Holtek é uma marca registrada da Holtek Semiconductor Inc.

Futaba é uma marca registrada da Futaba Corporation of America.

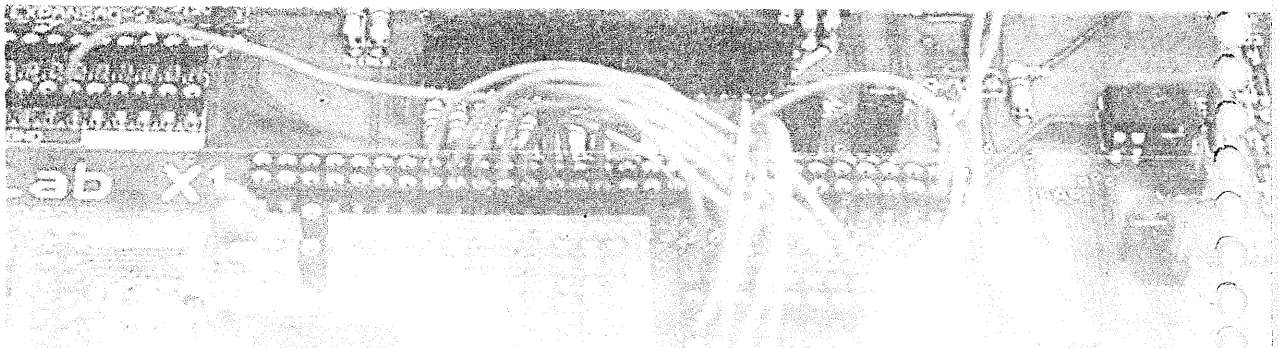
Hitec é uma marca registrada da Hitec RCD USA, Inc.

JR é uma marca registrada da Horizon Hobby, Inc.

Hobbico é uma cara registrada da Hobbico, Inc.

As figuras do capítulo 5 são baseadas nos originais da Texas Instruments e foram reproduzidas mediante autorização.

Todos os demais nomes registrados, marcas registradas ou direitos de uso citados neste livro pertencem aos seus respectivos proprietários.



Microcontroladores família MSP430

Teoria e Prática

Aborda a maioria dos periféricos disponíveis nas famílias 1xx, 2xx e 4xx de microcontroladores MSP 430 da Texas Instruments, além da arquitetura interna da CPU de 16 bits, modos de funcionamento e instruções Assembly.

O ambiente de programação Embedded Workbench da IAR é estudado, assim como as técnicas de simulação e depuração disponíveis nos chips. Aborda as características particulares do compilador C IAR, além de uma breve revisão da linguagem C no padrão ANSI.

Para auxiliar o aprendizado, apresenta diversos exemplos de configuração e utilização dos periféricos internos. Há um capítulo que visa explorar as características particulares dos chips e demonstrar a sua utilização, com exemplos de aplicação utilizando módulos LCD de caractere e gráfico, utilização de integrados SPI, teclados, comunicação serial, etc.

O livro possui uma linguagem clara e objetiva e pode ser utilizado por estudantes, profissionais e aficionados da área.

INVISTA EM VOCÊ. LEIA LIVROS!

www.editoraerica.com.br

ISBN: 85-365-0067-0

9 788536 500676