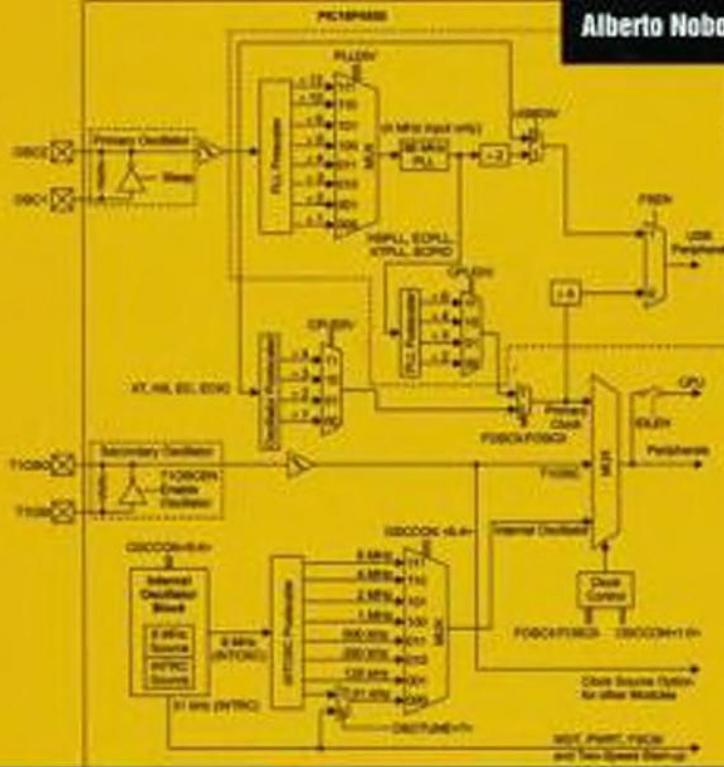


Alberto Noboru Miyadaira



Microcontroladores PIC18

Aprenda e Programe em **Linguagem C**

BRUNO

Respostas das questões de Física, Química, Biologia e Matemática disponíveis no site www.autodidata.com.br para download.



djvu AH12

Microcontroladores PIC18

Aprenda e Programe em Linguagem C



EDITORIA AFILIADA

Seja Nossa Parceiro no Combate à Cópia Ilegal

A cópia ilegal é crime. Ao efetuá-la, o infrator estará cometendo um grave erro, que é inibir a produção de obras literárias, prejudicando profissionais que serão atingidos pelo crime praticado.

Junte-se a nós nesta corrente contra a pirataria. Diga não à cópia ilegal

Seu Cadastro É Muito Importante para Nós

Se você não comprou o livro pela Internet, ao preencher a ficha de cadastro em nosso site, você passará a receber informações sobre nossos lançamentos em sua área de preferência.

Conhecendo melhor nossos leitores e suas preferências, vamos produzir títulos que atendam suas necessidades.

Obrigado pela sua escolha.

Fale Conosco!

Eventuais problemas referentes ao conteúdo deste livro serão encaminhados ao(s) respectivo(s) autor(es) para esclarecimento, excetuando-se as dúvidas que dizem respeito a pacotes de softwares, as quais sugerimos que sejam encaminhadas aos distribuidores e revendedores desses produtos, que estão habilitados a prestar todos os esclarecimentos.

Os problemas só podem ser enviados por:

1. E-mail: producao@erica.com.br
2. Fax: (11) 2097.4060
3. Carta: Rua São Gil, 159 - Tatuapé - CEP 03401-030 - São Paulo - SP



Alberto Noboru Miyadaira

**Microcontroladores PIC18
Aprenda e Programe em Linguagem C**

1^a Edição

**São Paulo
2009 - Editora Érica Ltda.**

Copyright © 2009 da Editora Érica Ltda.

Todos os direitos reservados. Proibida a reprodução total ou parcial, por qualquer meio ou processo, especialmente por sistemas gráficos, microfilmicos, 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 edição. 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 19.06.1998, 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 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 para download no site da Editora Érica.

Conteúdo adaptado ao Novo Acordo Ortográfico da Língua Portuguesa, em execução desde 1º de janeiro de 2009.

"Algumas imagens utilizadas neste livro foram obtidas a partir do CorelDRAW 12, X3 e X4 e da Coleção do MasterClips/MasterPhotos da IMSI, 100 Rowland Way, 3rd floor Novato, CA 94945, USA"

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

Miyadaira, Alberto Noboru

Microcontroladores PIC18. aprenda e programe em linguagem C / Alberto Noboru Miyadaira. -- 1. ed. -- São Paulo: Érica, 2009.

Bibliografia.

ISBN 978-85-365-0244-1

1. C (Linguagem de programação para computadores) 2. Microcontroladores
 I. Título.

09-06845

CDD-005.133

Índices para catálogo sistemático

1. C: Linguagem de programação: Aplicações com microcontroladores PIC18:

Computadores: Processamento de dados 005.133

Conselho Editorial:

Revisão e Coordenação Editorial: Rosana Arruda da Silva

Avaliador Técnico: Fábio Pereira

Capa: Mauricio S. de França

Editoração: Adriana Aguiar Santoro

Desenhos: Flávio Eugenio de Lima

Revisão: Mariene Teresa S. Alves

Carla de Oliveira Moraes

Editora Érica Ltda.

Rua São Gil, 159 - Tatuapé

CEP: 03401-030 - São Paulo - SP

Fone: (11) 2295-3066 - Fax: (11) 2097-4060

www.editoraerica.com.br

Dedicatória

Aos meus pais Nelson Toshikazu Miyadaira e Natsue Kyosen Miyadaira;

Aos meus irmãos Cristina Miyadaira e Fernando Yukio Miyadaira, que tanto me apoiaram na realização deste trabalho;

À minha namorada Celenir Teló pelo apoio e compreensão quanto à importância deste livro como realização pessoal.

Agradecimentos

Escrever este livro foi uma experiência ímpar devido ao alto grau de complexidade deste trabalho. Significa superação, pois muitos desafios foram enfrentados e vencidos, com muita garra e determinação. Isso prova que não há limites para o potencial humano desde que esteja motivado e determinado.

Gostaria de agradecer a Artimar, representante exclusivo da Microchip Technology no Brasil, pelas amostras de alguns modelos de microcontroladores PIC®;

À equipe da Editora Érica por todo empenho e dedicação, especialmente à coordenadora Rosana Arruda da Silva, à responsável pela editoração do livro Adriana Aguiar Santoro e ao responsável pela capa Maurício Scervianinas de França;

A todos os amigos que me ajudaram, pois sem seu apoio e o de minha família teria sido muito difícil finalizar esta obra;

Ao leitor pelo voto de confiança.

"Bem sei eu que tudo podes e nenhum dos teus pensamentos pode ser impedido."

Jó 42:1

Fabricantes

Microchip Technology Inc.

Produtos: microcontroladores PIC®

Site: www.microchip.com

Representante oficial no Brasil:

Artimar Ltda.

Rua Bela Cintra 746 - 3º piso

CEP: 01415-000 - São Paulo-SP

e: (11) 3231-0277

x: (11) 3255-0511

www.artimar.com.br

Requisitos de Hardware e de Software

Software

Sistema operacional Windows 2000, XP ou superior;

Ambiente de desenvolvimento MPLAB® IDE;

Compilador MPLAB® C18.

Hardware

Processador com clock superior a 600 MHz;

128 MB de memória RAM;

Disco rígido de 400 Mbytes (MB);

Unidade de DVD-ROM;

Acesso à Internet;

Monitor de vídeo configurado com no mínimo 800x600 pontos.

Índice Analítico

Capítulo 1 - Introdução	21
Exercícios.....	23
Capítulo 2 - MPLAB® IDE	24
2.1 Criação de um Projeto	24
2.2 Adição de Arquivo ao Projeto	25
2.3 Configuração do Compilador MPLAB® C18.....	26
2.4 Configuração do Microcontrolador	27
2.4.1 Select Device...	27
2.4.2 Configuration Bits.....	28
2.5 Compilação do Projeto	28
2.6 Verificação da Quantidade de Memória de Dados e de Programa Utilizada pelo Código	29
2.7 Visualização e Alteração do Conteúdo da Memória EEPROM Interna.....	30
2.8 Verificação e Alteração do Conteúdo dos Registradores e Variáveis do Projeto	30
2.9 Visualização das Variáveis Locais	31
2.10 Simulação e Depuração do Código-Fonte do Projeto	31
2.11 Simulação de UART pelo MPLAB® SIM	32
2.12 Gravação do Programa no Microcontrolador	33
Capítulo 3 - Compilador MPLAB® C18.....	35
3.1 Considerações Iniciais	35
3.2 Linha de Comando	35
3.2.1 Opções de Otimização	37
3.2.1.1 Dead Code Removal	38
3.2.1.2 Integer Promotion	39
3.2.1.3 Duplicate String Merging	39
3.2.1.4 Banking	39
3.2.1.5 Copy Propagation	40
3.2.1.6 Redundant Store Removal	40
3.2.1.7 Unreachable Code Removal	41
3.2.1.8 Tail Merging	41
3.2.1.9 Branches	41
3.2.1.10 Code Straightening	42
3.2.1.11 Wreg Content Tracking	42
3.2.1.12 Procedural Abstraction	43
3.3 Modo Estendido (Extended Mode)	43
Capítulo 4 - Linguagem C MPLAB® C18.....	44
4.1 Comentários	44
4.2 Identificadores	44
4.3 Palavras-Chaves	45
4.4 Tipos de Dados	45
4.5 Tipos de Qualificadores	46
4.5.1 Qualificadores de Armazenamento	46
4.5.1.1 Memória de Dados	46
4.5.1.2 Memória de Programa	46
4.5.1.3 Ponteiros	47

4.5.2 Classes de Armazenamento	47
6 Instrução	48
7 Declaração	49
8 Representações dos Dados Numéricos	49
9 Matrizes	50
4.9.1 Unidimensional	50
4.9.2 Multidimensional	51
10 Operadores	51
4.10.1 Atribuição	51
4.10.2 Aritiméticos	52
4.10.3 Bit a Bit	52
4.10.4 Relacionais	53
4.10.5 Lógicos	54
4.10.6 Ponteiros	54
11 Funções	55
12 Comandos de Seleção	57
4.12.1 Comando If	57
4.12.2 Comando Switch	57
13 Laços	59
4.13.1 Laço While	59
4.13.2 Laço Do-While	59
4.13.3 Laço For	60
14 Comandos de Desvio	61
4.14.1 Comando Break	61
4.14.2 Comando Continue	61
4.14.3 Comando Goto	62
4.14.4 Comando Return	62
15 Enumerações, Estruturas, Tipos de Dados Definidos pelo Usuário e Uniões	63
4.15.1 Enumerações	63
4.15.2 Estruturas	64
4.15.3 Tipos de Dados Definidos pelo Usuário	65
4.15.4 Uniões	66
16 Diretivas Básicas	66
4.16.1 #Define e #Undef	67
4.16.2 #Error	67
4.16.3 #If	68
4.16.4 #Ifdef e #Ifndef	68
4.16.5 #Include	69
4.16.6 #Line	70
4.16.7 #Pragma	70
4.16.7.1 #Pragma Sectiontype	70
4.16.7.2 #Pragma Tmpdata	72
4.16.7.3 #Pragma Varlocate	73
17 Macros Predefinidas	74
4.17.1 __DATE__	74
4.17.2 __FILE__	74
4.17.3 __LINE__	74
4.17.4 __TIME__	75
4.17.5 __STDC__	75
4.17.6 __18CXX__	75

4.17.7 _Nameprocessor	75
4.17.8 _SMALL	76
4.17.9 _LARGE	76
4.17.10 _TRADITIONAL18	76
4.17.11 _EXTENDED18	76
4.18 Funções de Saída de Caracteres	76
4.18.1 Putc	77
4.18.2 Puts	77
4.18.3 Fputs	78
4.18.4 Printf	78
4.18.5 Fprintf	81
4.18.6 Sprintf	81
4.18.7 Vprintf	81
4.18.8 Vfprintf	82
4.18.9 Vsprintf	83
4.18.10 _Usart_Putc	83
4.18.11 _User_Putc	83
4.19 Funções Diversas	84
4.19.1 Funções de Manipulações de Bit/Byte	84
4.19.1.1 RLNCF E RRNCF	84
4.19.1.2 RLCF E RRRCF	85
4.19.1.3 SWAPF	86
4.19.2 Funções de Classificação de Caracteres	87
4.19.2.1 Isalnum	87
4.19.2.2 Isalpha	87
4.19.2.3 Isdigit	88
4.19.2.4 Islower	88
4.19.2.5 Isspace	89
4.19.2.6 Isupper	89
4.19.2.7 Isxdigit	89
4.19.2.8 Tolower	90
4.19.2.9 Toupper	90
4.19.2.10 Iscntrl	90
4.19.2.11 Isgraph	91
4.19.2.12 Isprint	91
4.19.2.13 Ispunct	92
4.19.3 Funções de Conversão de Dados	92
4.19.3.1 Atob, Atof, Atoi e Atol	92
4.19.3.2 Btoa, Itoa, Ltoa e Ulttoa	93
4.19.4 Funções de Manipulação de Memória e String	94
4.19.4.1 Memchr e Memchrpgm	94
4.19.4.2 Memcmp, Memcmppgm, Memcmppgm2ram e Memcmppram2pgm	95
4.19.4.3 Memcpy, Memcpypgm, Memcpypgm2ram e Memcpypiram2pgm	95
4.19.4.4 Memmove, Memmovepgm, Memmovepgm2ram e Memmoveveram2pgm	96
4.19.4.5 Memset e Memsetpgm	97
4.19.4.6 Strcat, Strcatpgm, Strcatpgm2ram e Strcatram2pgm	98
4.19.4.7 Strchr e Strchrpgm	99
4.19.4.8 Strcmp, Strcmppgm, Strcmppgm2ram e Strcmpram2pgm	100
4.19.4.9 Strcpy, Strcpypgm, Strcpypgm2ram e Strcpypiram2pgm	101
4.19.4.10 Strcsn, Strcsnpgm, Strcsnpgmram e Strcsnprampgm	102

4.19.4.11 Strlen e Strlengm	102
4.19.4.12 Strlwr e Strlwrgm	103
4.19.4.13 Strncat, Strncatpgm, Strncatpgm2ram e Strncatram2pgm	104
4.19.4.14 Strncmp, Strncmppgm, Strncmppgm2ram e Strncmpram2pgm	104
4.19.4.15 Strncpy, Strncpypgm. Strncpypgm2ram e Strncpyram2pgm	106
4.19.4.16 Strpbrk, Strpbrkpgm, Strpbrkpgmram e Strpbrkrampgm	106
4.19.4.17 Strrchr	107
4.19.4.18 Strspn, Strspnpgm, Strspnpgmram e Strspnrampgm	108
4.19.4.19 Strstr, Strstrpgm, Strstrpgmram e Strstrrampgm	109
4.19.4.20 Strtok, Strtokpgm, Strtokpgmram e Strtokrampgm	110
4.19.4.21 Strupr e Struprpgm	111
4.19.5 Funcões Matemáticas	112
4.19.5.1 Acos, Asin, Atan e Atan2	112
4.19.5.2 Ceil e Floor	113
4.19.5.3 Cos, Sin e Tan	114
4.19.5.4 Cosh, Sinh e Tanh	114
4.19.5.5 Exp	115
4.19.5.6 Fabs	115
4.19.5.7 Fmod	115
4.19.5.8 Frexp	116
4.19.5.9 Ieeeatomchp e Mchptoiieee	116
4.19.5.10 Ldexp	117
4.19.5.11 Log e Log10	117
4.19.5.12 Modf	118
4.19.5.13 Pow	118
4.19.5.14 Sqrt	119
4.19.6 Números Pseudoaleatórios	119
4.19.6.1 Rand	119
4.19.6.2 Strand	120
Código em Assembly	120
Funções de Controle do Processador	129
21.1 Clwdt	129
4.21.2 Descrição dos Resets	129
4.21.2.1 isBOR ()	129
4.21.2.2 isLVD ()	129
4.21.2.3 isMCLR ()	129
4.21.2.4 isPOR ()	130
4.21.2.5 isWDTTO ()	130
4.21.2.6 isWDTWU ()	130
4.21.2.7 isWU ()	130
4.21.3 Funções de Atraso	130
4.21.4 Nop	131
4.21.5 Reset	131
4.21.6 Sleep	132
Arquivos do Autor	132
4.22.1 Memória EEPROM Interna	132
4.22.1.1 escreve_mem_EEPROM ()	132
4.22.1.2 le_mem_EEPROM ()	132
4.22.2 Memória Flash Interna	133
4.22.2.1 escreve_mem_flash ()	133

4.22.2.2 le_mem_flash ()	133
4.23 Dicas	134
Exercícios	135
Capítulo 5 - Microcontrolador PIC18F4550	137
5.1 Introdução	137
5.1.1 Memórias	137
5.1.2 Ciclo de Máquina	138
5.2 Pinagem	139
5.3 Diagrama de Blocos do PIC18F4550	142
5.4 Memória de Dados	144
5.4.1 Registradores de Funções Especiais (SFRs)	146
5.5 Memória de Programa e a Stack	150
5.5.1 Memória de Programa	150
5.5.2 Vetores	151
5.5.3 Stack (Pilha)	151
5.5.4 Verificação e Proteção do Código do Programa	152
5.6 Oscilador	152
5.6.1 Oscilador Interno	153
5.6.1.1 Modos do Oscilador Interno	154
5.6.2 Oscilador Secundário	154
5.6.3 Oscilador Primário	155
5.6.3.1 Configuração do Oscilador Externo	156
5.6.4 Funções do Oscilador para a USB	157
5.7 Gerenciamento de Energia	157
5.8 Reset	157
5.8.1 Fonte de Reset	158
5.8.1.1 Eventos Internos	159
5.8.1.2 Evento Externo	159
5.8.2 Contadores de Reset do Dispositivo	160
5.8.3 Two-Speed Start-Up	161
5.9 Características Elétricas do PIC18F4550	161
5.10 Fonte de Alimentação	161
5.11 Frequência x Tensão de Alimentação	162
5.12 Funções Diversas do PIC18	162
5.12.1 Registrador de Status	162
5.12.2 Fail-Safe Clock Monitor (FSCM)	163
5.12.3 Instruções Estendidas	163
5.12.4 High/Low-Voltage Detect (HLVD)	163
5.13 Métodos de Programação	164
5.14 Tipos de Encapsulamento	165
5.15 Identificação do Microcontrolador PIC®	165
5.16 Arquivo de Cabeçalho	166
Exercícios	167
Capítulo 6 - Configuração do PIC18	168
Capítulo 7 - Portas I/O Digitais	174
7.1 Sentido do Fluxo de Dados da Porta	175
7.1.1 TRISA, TRISB, TRISC, TRISD e TRISE	175
7.1.2 TRISAbits, TRISBbits, TRISCbits, TRISDbits e TRISEbits	176

7.2 Controle do Estado dos Pinos da Porta.....	176
7.2.1 PORTA, PORTB, PORTC, PORTD e PORTE	176
7.2.2 PORTAbits, PORTBbits, PORTCbits, PORTDbits e PORTEbits	177
7.3 Registro LAT	177
7.3.1 LATA, LATB, LATC, LATD e LATE	178
7.3.2 LATAbits, LATBbits, LATCbits, LATDbits e LATEbits	179
7.4 Habilita/Desabilita Pull-Ups Internos	179
Exercícios	180
7.5 Projeto	181
Capítulo 8 - Display LCD 2X16	183
8.1 Pinagem LCD 2x16	183
8.2 Instruções de Controle	184
8.2.1 Configuração do Cursor e do Display	184
8.2.2 Controle do Display/Cursor	185
8.2.3 Controle da Mensagem	185
8.2.4 Status e Posição do Contador de Endereço	186
8.2.5 Leitura e Escrita de Dados	186
8.2.6 Endereço da Linha x Coluna	186
8.2.7 Caractere Especial	187
8.3 Inicialização do Display LCD 2x16 com Oito Vias	188
8.4 Inicialização do Display LCD 2x16 com Quatro Vias	188
8.5 Conjuntos de Caracteres do Display	189
8.6 Biblioteca do Display LCD Alfanumérico	190
8.7 Projeto	191
Capítulo 9 - Interrupção	195
9.1 Bits de Configuração da Interrupção	196
9.1.1 Interrupção com Nível de Prioridade	196
9.1.2 Interrupção Sem Nível de Prioridade	197
9.1.3 Bits de Configuração do Evento de Interrupção	197
9.2 Comportamento da Interrupção	201
9.3 Diretiva de Interrupção	203
9.4 Período de Latência	205
9.5 Projeto	206
Capítulo 10 - USART	209
10.1 Protocolo RS-232	209
10.1.1 Funcionamento do Protocolo RS-232	210
10.1.2 Níveis Lógicos da Interface RS-232	211
10.2 Módulo EUSART do PIC18F4550	211
10.2.1 Funções Adicionais da EUSART	211
10.2.1.1 Wake-Up Automático na Recepção de Dado	211
10.2.1.2 Autodetecção e Calibração do Baud Rate	212
10.2.1.3 Transmissão de Caractere Break de 12bits	213
10.2.1.4 Seleção da Polaridade do Clock	213
10.3 Funções de Configuração	213
10.3.1 Desabilita USART	213
10.3.2 Habilita USART	213
10.3.3 Bits de Configuração do Baud Rate da EUSART	215

10.4 Funções de Controle	216
10.4.1 Status da Recepção	216
10.4.2 Status da Transmissão	216
10.4.3 Transmissão de Caractere	217
10.4.4 Transmissão de String	218
10.4.4.1 Dados Localizados na Memória de Dados	218
10.4.4.2 Dados Localizados na Memória de Programa	219
10.4.5 Recepção de Caractere	219
10.4.6 Recepção de String	219
10.5 Funções UART Implementadas em Software	221
10.5.1 Definição das Funções de Atraso	221
10.5.2 Configuração da UART em Software	222
10.5.3 Transmissão de Caractere	222
10.5.4 Transmissão de String	222
10.5.5 Recepção de Caractere	223
10.5.6 Recepção de String	223
10.6 Projeto	225
Capítulo 11 - TIMERS e Watchdog Timer (WDT)	228
11.1 TIMERS	228
11.1.1 Características dos TIMERS do PIC18F4550	229
11.1.1.1 TIMER 0	229
11.1.1.2 TIMER 1	229
11.1.1.3 TIMER 2	229
11.1.1.4 TIMER 3	229
11.1.2 Funções de TIMER	230
11.1.2.1 Desabilita TIMER	230
11.1.2.2 Habilita TIMER 0	230
11.1.2.3 Habilita TIMER 1	231
11.1.2.4 Habilita TIMER 2	231
11.1.2.5 Habilita TIMER 3	232
11.1.2.6 Habilita TIMER 4	233
11.1.2.7 Operação de Leitura	233
11.1.2.8 Operação de Escrita	234
11.1.2.9 Seleção do TIMER para o Módulo CCP	234
11.2 Watchdog Timer (WDT)	235
11.2.1 Função de Reinício do Contador de WDT	235
11.3 Projeto	237
Capítulo 12 - Módulo CCP/ECCP	243
12.1 Módulo CCP/ECCP do PIC18F4550	243
12.1.1 Modo PWM	243
12.1.2 Módulo ECCP	245
12.1.2.1 Modo PWM com Capacidade Aumentada	246
12.2 Funções para o Módulo CCP/ECCP	252
12.2.1 Funções do Modo Capture	252
12.2.1.1 Desabilita o Capture	252
12.2.1.2 Habilita o Capture	252
12.2.1.3 Operação de Leitura do Capture	253
12.2.2 Funções do Modo Compare	254
12.2.2.1 Desabilita o Compare	254

12.2.2.2 Habilita o Compare	254
12.2.3 Funções para o Modo PWM	255
12.2.3.1 Desabilita o PWM	255
12.2.3.2 Habilita o PWM	255
12.2.3.3 Seta o Duty Cycle do Sinal PWM	256
12.2.3.4 Define a Saída de PWM do Módulo ECCP	257
Projetos	258
3.1 Capture	259
3.2 Compare	261
3.3 PWM	263
13 - Conversor Analógico-Digital	266
Conversor A/D do PIC18F4550	266
3.1.1 Tempo de Aquisição e Conversão do Sinal	267
3.1.2 Conversão do Sinal Analógico	269
Operações para o Módulo Conversor A/D	270
3.1 Verifica o Estado do Módulo	270
3.2 Desabilita o Módulo	270
3.3 Inicia a Conversão A/D	270
3.4 Habilita o Módulo Conversor A/D	271
3.2.5 Operação de Leitura	277
3.2.6 Seleção do Canal Analógico	277
Projetos	277
3.3.1 Leitura da Tensão Regulada por um Potenciômetro e Sensor de Temperatura	278
3.3.2 Teclado Analógico	281
14 - Módulo Comparador Analógico e de Tensão de Referência	285
Módulo Comparador	285
Módulo de Tensão de Referência	287
Projeto	289
15 - Comunicação I²C	291
Funcionamento do Protocolo I ² C	292
Comunicação I ² C do PIC18F4550	293
15.2.1 Registro de Endereço/Baud Rate (SSPADD)	293
Funções de Controle/Configuração do Periférico I ² C	294
15.3.1 Condição de Acknowledge (ACK)	294
15.3.2 Condição de Not Acknowledge (Not ACK)	294
15.3.3 Condição de RESTART	294
15.3.4 Condição de START	294
15.3.5 Condição de STOP	295
15.3.6 Configura o Periférico I ² C	295
15.3.7 Desabilita o Periférico I ² C	296
15.3.8 Recepção de Caractere	296
15.3.9 Recepção de String	296
15.3.10 Status do Barramento I ² C	297
15.3.11 Status do Buffer de Recepção	297
15.3.12 Transmissão de Caractere	297
15.3.13 Transmissão de String	298
Funções I ² C Implementadas em Software	298

15.4.1 Alongamento do Clock para o Modo Slave	299
15.4.2 Condição de Acknowledge (ACK)	300
15.4.3 Condição de Not Acknowledge (Not ACK)	300
15.4.4 Condição de RESTART	300
15.4.5 Condição de START	300
15.4.6 Condição de STOP	300
15.4.7 Recepção de Caractere	301
15.4.8 Recepção de String	301
15.4.9 Transmissão de Caractere	301
15.4.10 Transmissão de String	302
15.5 Projeto	302
15.5.1 Memória EEPROM 24C128	302
15.5.2 Modo de Funcionamento	303
15.5.3 Circuito Eletrônico Proposto para o Projeto	304
Capítulo 16 - Comunicação SPI.....	308
16.1 Funcionamento do Protocolo SPI.....	309
16.2 Comunicação SPI do PIC18F4550.....	310
16.3 Funções de Controle/Configuração do Periférico SPI.....	310
16.3.1 Configura o Periférico SPI.....	311
16.3.2 Desabilita o Periférico SPI.....	311
16.3.3 Recepção de Caractere	312
16.3.4 Recepção de String	312
16.3.5 Status do Buffer de Recepção	312
16.3.6 Transmissão de Caractere	313
16.3.7 Transmissão de String	313
16.4 Funções SPI Implementadas em Software	314
16.4.1 Configura os Pinos I/O	315
16.4.2 Transmissão de Caractere	315
16.4.3 Limpa o Pino Chip Select (\overline{CS})	315
16.4.4 Seta o Pino Chip Select (\overline{CS})	315
16.5 Exemplo	316
Capítulo 17 - SD Card	323
17.1 Organização da Memória	324
17.2 Registradores do SD Card	324
17.2.1 Registrador de Condição de Operação (OCR)	324
17.2.2 Registrador de Identificação do Cartão (CID)	325
17.2.3 Registrador de Dado Específico do Cartão (CSD)	326
17.2.4 Registrador das Configurações Especiais (SCR)	328
17.3 Pinagem da Memória SD Card	328
17.4 Modos de Instalação	329
17.5 Comandos Suportados pelo SD Card	330
17.5.1 Comandos Básicos	330
17.5.2 Comandos Específicos	331
17.6 Formato Padrão da Comunicação	332
17.7 Respostas dos Comandos	332
17.7.1 Resposta R1.....	333
17.7.2 Resposta R1b.....	333
17.7.3 Resposta R2.....	333

17.7.4 Resposta R3.....	334
17.8 Sinais Relacionados aos Dados	334
17.8.1 Sinal de Início e Parada de Transmissão.....	334
17.8.2 Sinal de Status da Escrita de Dado.....	334
17.8.3 Sinal de Erro de Dado	335
17.9 Operações para Ler e Escrever Dados	335
17.9.1 Operação de Leitura de Um Bloco	335
17.9.2 Operação de Leitura de Múltiplos Blocos.....	336
17.9.3 Operação de Escrita em Um Bloco	336
17.9.4 Operação de Escrita em Múltiplos Blocos.....	336
17.10 Inicialização do SD Card	337
17.11 Biblioteca do Cartão SD Card.....	338
17.12 Projeto	339
Capítulo 18 - USB (Universal Serial Bus)	344
18.1 Introdução	344
18.2 Topologia USB.....	344
18.3 Pinagem dos Conectores Padrão.....	345
18.4 Taxas de Transferência Suportadas pelo USB	346
18.5 Codificação/Decodificação NRZI	347
18.6 Endpoint e Pipe	347
18.7 Protocolo USB	348
18.7.1 Campo de Identificação do Pacote.....	349
18.7.2 Pacote Token	350
18.7.2.1 Campo ADDR	351
18.7.2.2 Campo ENDP	351
18.7.2.3 Campo CRC5	351
18.7.3 Pacote Data.....	351
18.7.4 Pacote de Handshake	353
18.7.4.1 Transação IN	353
18.7.4.2 Transação OUT	354
18.7.4.3 Transação SETUP	354
18.7.5 Pacote Special.....	354
18.8 Funcionamento dos Tipos de Transferência	354
18.8.1 Bulk Data Transfers.....	355
18.8.2 Control Transfers.....	355
18.8.2.1 Etapa de Setup	356
18.8.2.2 Etapa de Data	358
18.8.2.3 Etapa de Status	358
18.8.3 Isochronous Data Transfers	359
18.8.4 Interrupt Data Transfers	360
18.9 Recursos Padrão do Dispositivo USB	360
18.9.1 Device Remote Wakeup.....	360
18.9.2 Endpoint Halt.....	360
18.9.3 Test Mode.....	361
18.9.3.1 TEST_J	361
18.9.3.2 TEST_K	361
18.9.3.3 TEST_SE0_NAK	361
18.9.3.4 TEST_PACKET	361
18.9.3.5 TEST_FORCE_ENABLE	361

18.10 Descritores Padrão	362
18.10.1 Configuration Descriptor....	362
18.10.2 Device Descriptor	363
18.10.3 Device_Qualifier Descriptor.....	364
18.10.4 Endpoint Descriptor.....	365
18.10.5 Interface Descriptor	366
18.10.6 Other_Speed_Configuration Descriptor	367
18.10.7 String Descriptor.....	368
18.11 Classes	369
18.12 Requisições Padrão da USB	370
18.12.1 Clear_Feature	370
18.12.2 Get_Configuration	371
18.12.3 Get_Descriptor	371
18.12.4 Get_Interface	371
18.12.5 Get_Status	371
18.12.6 Set_Address	372
18.12.7 Set_Configuration.....	373
18.12.8 Set_Descriptor.....	373
18.12.9 Set_Feature	373
18.12.10 Set_Interface	374
18.12.11 Synch_Frame	374
18.13 Processo de Enumeração do Dispositivo	375
18.14 Características do Módulo USB do PIC18F4550	375
18.14.1 Serial Interface Engine (SIE).	377
18.14.2 Configuração do Oscilador para a SIE ..	377
18.15 Bibliotecas para a Comunicação USB.....	377
18.16 USB Hardware Abstraction Layer (HAL)	377
18.17 Configuração do Módulo USB	378
18.18 Arquivo de Descritores	380
18.19 Funções de Controle da USB.....	381
18.19.1 USBDeviceInit	381
18.19.2 USBDeviceTasks	381
18.19.3 USBEnableEndpoint.....	382
18.19.4 USBStallEndpoint.....	383
18.19.5 USBTransferOnePacket.....	383
18.19.6 USBDeviceDetach.....	384
18.19.7 USBDeviceAttach.....	384
18.20 Biblioteca USB CDC	384
18.20.1 USBCheckCDCRequest	386
18.20.2 CDCInitEP	386
18.20.3 getsUSBUSART	387
18.20.4 putUSBUSART	387
18.20.5 putsUSBUSART e putrsUSBUSART	388
18.20.6 CDCTxService	388
18.21 Projeto	389
Apêndice A - Tabela ASCII.....	395
Bibliografia	397
Índice Remissivo	399

Sobre o Material Disponível na Internet

O material disponível no site da Editora Érica contém as respostas dos exercícios, alguns códigos-fonte e as bibliotecas de manipulação das memórias EEPROM e FLASH interna do microcontrolador PIC18F4550, além do display LCD 2x16 e SD Card. Para abrir os arquivos disponíveis é necessário instalar na máquina os softwares Adobe Acrobat Reader 5.0 ou superior, MPLAB® C18 v3.32 e MPLAB® IDE v8.36 ou superior, que são estudados nesta obra e podem ser baixados do site oficial da Microchip Technology (www.microchip.com).

PIC18.exe - 479 KB

Procedimento para Download

Acesse o site da Editora Érica Ltda.: 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). Aparecem os dados do livro e o arquivo para download. Com um clique o arquivo executável é transferido.

Por meio do botão "Download". Na página principal do site, clique no item "Download". É exibido um campo no qual devem ser digitadas palavras-chave (nome do livro ou do autor). Aparecem o nome do livro e o arquivo para download. Com um clique o arquivo executável é transferido.

Procedimento para Descompactação

Primeiro passo: após ter transferido o arquivo, verifique o diretório em que se encontra e dê um duplo-clique nele. Aparece uma tela do programa WINZIP SELF-EXTRACTOR que conduz ao processo de descompactação. Abaixo do Unzip To Folder há um campo que indica o destino do arquivo que será copiado para o disco rígido do seu computador.

\PIC18

Segundo passo: prossiga a instalação, clicando no botão Unzip, o qual se encarrega de descompactar o arquivo. Logo abaixo dessa tela aparece a barra de status que monitora o processo para que você acompanhe. Após o término, outra tela de informação surge, indicando que o arquivo foi descompactado com sucesso e está no diretório criado. Para sair dessa tela, clique no botão OK. Para finalizar o programa WINZIP SELF-EXTRACTOR, clique no botão Close.

Prefácio

Esta obra transmite o conteúdo de forma simples e objetiva, enfatizando o rápido aprendizado e a autonomia, uma vez que oferece capítulos sequencialmente organizados, o que garante melhor compreensão dos recursos e das funcionalidades do PIC18F4550.

Os capítulos abordam a arquitetura e os métodos de programação em linguagem C, destinados à família PIC18, permitindo maior flexibilidade no desenvolvimento do programa.

Primeiramente apresenta um tutorial sobre o ambiente de desenvolvimento MPLAB® IDE v8.36, técnicas de programação em linguagem C com o compilador MPLAB® C18 v3 32, arquitetura e periféricos do PIC18F4550 e configurações.

A ordem dos capítulos foi estrategicamente organizada, de modo a propiciar melhor aproveitamento dos periféricos do modelo em estudo. Inicia com a manipulação de portas de entrada/saída digitais, display LCD 2x16, eventos de interrupção, comunicação com o PC via RS-232, Timer e Watchdog Timer, Real Timer Clock, temporizador, contador de eventos, geração de sinal PWM, conversor A/D, teclado analógico, medição de temperatura com o auxílio do sensor LM35, comparador analógico, acesso à EEPROM externa via interface I²C, acesso à memória SD Card via interface SPI e comunicação com uma porta COM virtual utilizando o módulo USB interno.

Traz uma lista com diversos exemplos e exercícios de fixação. Ao final dos capítulos descreve projetos de forma detalhada para melhor assimilação do conteúdo proposto.

Os projetos práticos podem ser realizados pela montagem dos circuitos eletrônicos proposta no livro ou pela obtenção de kits didáticos comercializados para o modelo PIC18F4550, modificando alguns trechos dos códigos-fonte.

Boa leitura!

O autor

Sobre o Autor

Alberto Noboru Miyadaira é formado em Matemática pelo método KUMON, Curso Técnico em Informática pelo SENAI (Serviço Nacional de Aprendizagem Industrial) e Engenharia de Controle e Automação pela FAG (Faculdade Assis Gurgacz). Durante a graduação estudou um ano na França, na ENSICAEN (*Ecole Nationale Supérieure d'Ingénieurs de Caen*) e após finalizar o período de experiência no exterior, fez uma pesquisa sobre "Teste Não Destruutivo por Correntes de Foucault" no laboratório de pesquisa GREYC (*Groupe de Recherche en Informatique, Image, Automatique et Instrumentation de Caen*), localizado na França. Trabalha com microcontroladores PIC® desde 2004 e a partir dessa época acompanha as novas tecnologias desenvolvidas pela Microchip Technology (fabricante de microcontroladores PIC®). É um aficionado desses pequenos dispositivos "inteligentes", por permitirem o desenvolvimento da capacidade inventiva. Atualmente cursa mestrado na UNICAMP em Engenharia Elétrica e realiza pesquisa na área de robótica.



Introdução

Os microcontroladores (μ C ou MCU) são pequenos dispositivos dotados de "inteligência", basicamente constituídos de CPU (Central Processing Unit em inglês, ou Unidade Central de Processamento em português), memória (dados e programas) e periféricos (portas E/S, I²C, SPI, USART etc.). Suas dimensões reduzidas são resultantes da alta capacidade de integração, em que milhões de componentes são inseridos em uma única pastilha de silício pela técnica de circuitos integrados (Cl's). Eles estão presentes na maioria dos equipamentos digitais, como celulares, MP3 player, impressoras, robótica, instrumentação, entre outros.

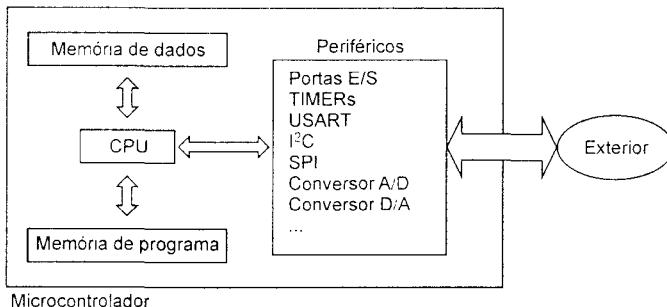


Figura 1.1. Diagrama resumido de um microcontrolador.

As duas principais arquiteturas de microcontroladores são *Harvard* e *Von-Neumann*. A arquitetura *Harvard* é caracterizada pela existência de um barramento para o acesso à memória de dados e outro para a memória de programa, resultando em um aumento de fluxo de dados, enquanto na arquitetura *Von-Neumann* as memórias de dados e de programa compartilham o mesmo barramento, limitando a banda de operação.

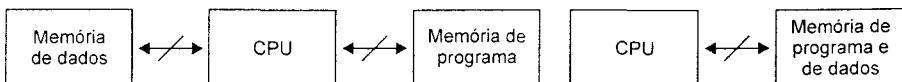


Figura 1.2: Diagrama da arquitetura *Harvard* (à esquerda) e arquitetura *Von-Neumann* (à direita).

Em geral, as memórias de programa presentes nos microcontroladores são do tipo FLASH (*Electrically Erasable Programmable Read Only Memory*), ROM (*Read Only Memory*), EPROM (*Erasable Programmable Read Only Memory*) ou OTP (*One Time Programmable*). Elas são responsáveis pelo armazenamento do programa, o que significa que sua capacidade de armazenamento deve ser suficiente para reter todo o código desejado. Essas memórias são do tipo não volátil, portanto o código de programa armazenado não é perdido, caso o circuito não esteja sendo alimentado.

Vejamos as principais características das memórias supracitadas.

- **ROM:** não permite que o conteúdo seja alterado pelo usuário. Ela aceita somente a leitura do conteúdo, o qual foi gravado pelo fabricante. Microcontroladores dotados de memória de programa do tipo ROM normalmente apresentam um baixo custo com relação às memórias FLASH, EPROM e OTP, e são recomendados quando o código do programa não apresenta erros e há necessidade de grande quantidade.
- **EPROM:** pode ser apagada e/ou programada muitas vezes, porém o conteúdo da memória é apagado através da exposição da janela de quartzo à luz ultravioleta, cujo processo de fabricação apresenta um custo elevado, se comparado com os outros tipos de memória.
- **OTP:** tolera somente uma gravação. Esse tipo de memória apresenta o menor custo se for comparado com as memórias programáveis (EPROM e FLASH).
- **FLASH:** é a memória mais flexível entre todos os outros tipos de memória de programa, pois pode ser apagada eletricamente e reprogramada 100.000 a 1.000.000 de vezes, dependendo da tecnologia empregada na fabricação desse componente.

Outro tipo de memória existente é a de dados, definida como memória RAM (*Random Access Memory*). Ela é volátil e armazena as variáveis e constantes do sistema. O conteúdo presente nesse tipo de memória é perdido sempre que a alimentação é cortada. Isso implica que os valores das variáveis devem ser carregados sempre que o sistema for iniciado.

Pinos I/O (*input/output* em inglês, ou entrada/saída em português) digitais estão presentes em todos os microcontroladores. Por meio deles o MCU se comunica com o mundo exterior, ou seja, é por intermédio destes que o MCU aciona um relé, lâmpada, motor etc. O sentido do fluxo de dados de um pino I/O pode ser definido como entrada ou saída. Se o pino for definido como saída, então ele normalmente será utilizado para controlar periféricos; caso contrário, se for definido como entrada, o dispositivo passa a ler o sinal presente no pino. Chamamos de porta um conjunto de pinos relacionados a ela (exemplo: porta X → pino x.1, x.2, x.3, x.4, x.5 etc.).

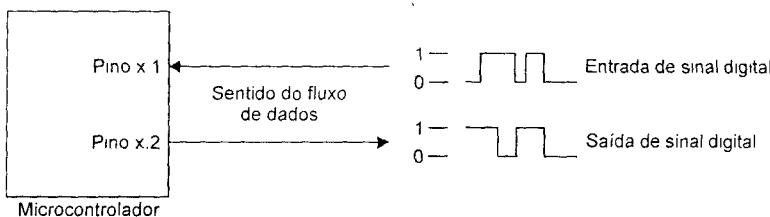


Figura 1.3: Fluxo de dados nos pinos de I/O digitais

Suponha que ambos os pinos x.1 e x.2 sejam do tipo I/O digital. Se o pino x.1 estiver configurado como entrada e o pino x.2 como saída, teremos o fluxo de dados representado pela Figura 1.3.

Alguns periféricos como conversores A/D, USART (*Universal Synchronous Asynchronous Receiver Transmitter*), TIMER, SPI (*Serial Peripheral Interface*) e I²C (*Inter-Integrated Circuit*) são muito comuns nos microcontroladores, no entanto existem MCUs mais robustos que, além dos periféricos supracitados, também apresentam outros mais específicos, como controladores de LCD, USB (*Universal Serial Bus*), RTC (*Real-Time Clock*), CAN etc.

A velocidade de processamento do microcontrolador está diretamente relacionada com a frequência de *clock*. Quanto maior a frequência de trabalho maior será a capacidade de processamento, assim como o consumo de energia. Essa frequência pode ser gerada por um oscilador interno, que normalmente é um circuito RC, ou então por um cristal de quartzo ou um ressonador conectado externamente. Osciladores internos do tipo RC são normalmente utilizados quando não há uma necessidade de precisão de *clock*; caso

contrário, utiliza-se o cristal. Por exemplo, se uma comunicação serial for necessária, é desejável que um cristal de quartzo seja empregado para garantir a fidelidade do sinal de *clock*, resultando maior confiabilidade na transmissão/recepção dos dados.

Exercícios

1. Qual é a memória designada ao armazenamento do conteúdo das variáveis e constantes do sistema?
 - a) Memória ROM.
 - b) Memória FLASH.
 - c) Memória RAM.
 - d) Memória OTP.
 - e) N.d.a.
2. Qual é o tipo de memória de programa que permite apagar e escrever milhares de vezes?
 - a) Memória FLASH.
 - b) Memória RAM.
 - c) Memória OTP.
 - d) N.d.a.
3. Qual é o significado de memória volátil?
 - a) É um tipo de memória cujo conteúdo é sempre mantido independente de qualquer evento.
 - b) É uma variedade de memória cujo conteúdo armazenado é perdido toda vez que a alimentação é retirada
 - c) É uma memória FLASH.
 - d) N.d.a.
4. Qual é a principal função de um cristal de quartzo?
 - a) Fornecer sinal de *clock* de baixa fidelidade.
 - b) Fornecer tensão de alimentação
 - c) Fornecer sinal de *clock* de alta fidelidade.
 - d) Estabilizar a corrente no circuito
 - e) N.d.a.



MPLAB® IDE

O MPLAB® IDE é um ambiente de desenvolvimento integrado, que permite fácil integração com o compilador MPLAB® C18 ou outro tipo de compilador, além de ser uma ferramenta extremamente poderosa, pois ela gerencia o projeto, compila, simula, debuga e grava o chip. Esse software pode ser baixado através do site oficial da Microchip Technology (www.microchip.com).

Com a finalidade de facilitar o aprendizado dessa ferramenta, segue um pequeno tutorial repleto de ilustrações, que vão auxiliar o leitor a familiarizar-se rapidamente com esse ambiente de desenvolvimento (MPLAB® IDE v8.36).

2.1 Criação de um Projeto

Clique em Project → Project Wizard.

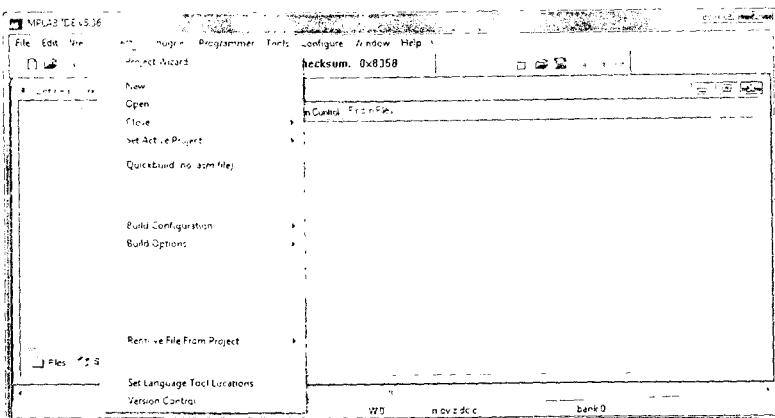


Figura 2.1

Uma tela de boas-vindas é aberta (*Welcome!*). Clique em **Next**.

Selecione o modelo do microcontrolador na barra "Device" e clique em **Next**, Figura 2.2.

Selecione o compilador na barra **Active Toolsuite** e clique em **Next**, Figura 2.3.

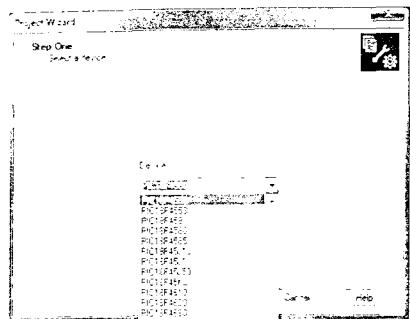


Figura 2.2

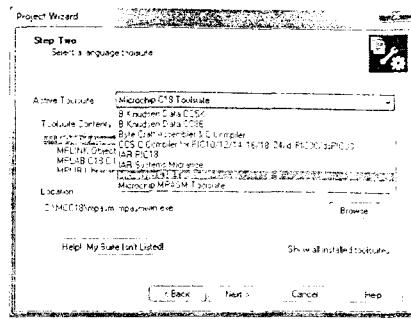


Figura 2.3



Os códigos presentes neste livro são compilados com o MPLAB® C18. Selecione e verifique se a localização do compilador está correta.

Clique em **Browse** para informar em qual pasta o projeto será armazenado, bem como o nome do projeto.

Clique duas vezes em **Next** e finalize clicando em **Finish**.

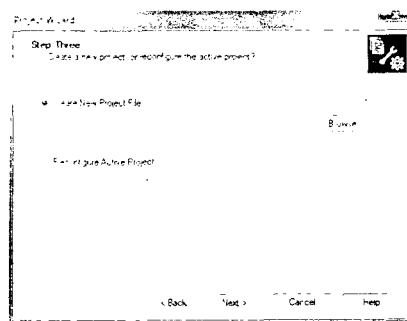


Figura 2.4

2.2 Adição de Arquivo ao Projeto

Para inserir um novo arquivo, clique em **File** → **New**. O arquivo *Untitled* é criado e o código-fonte do programa é escrito nesse arquivo.

Clique em **File** → **Save as**, informe o nome do arquivo com extensão '.c' e salve na pasta do projeto (exemplo: d:\Projeto\Meu primeiro projeto.c). Para adicioná-lo ao programa, clique com o botão direito do mouse no campo **Source Files** e adicione o arquivo (Add Files...) recentemente salvo, ou outro se desejar. Para abri-lo, clique duas vezes nele.



Após salvar o arquivo com o nome desejado, neste primeiro momento, copie e cole o código do arquivo teste.txt (pasta baixada do site da Editora Érica) no arquivo recém-criado. De acordo com o recomendado, essa pasta deve estar localizada em c:\pic18. Também é necessário inserir os arquivos **p18f4550.lib**, **clib.lib** e **c018i.o** na pasta em que se encontra o projeto. Esses arquivos estão localizados em **C:\MCC18\lib** (Capítulo 3).

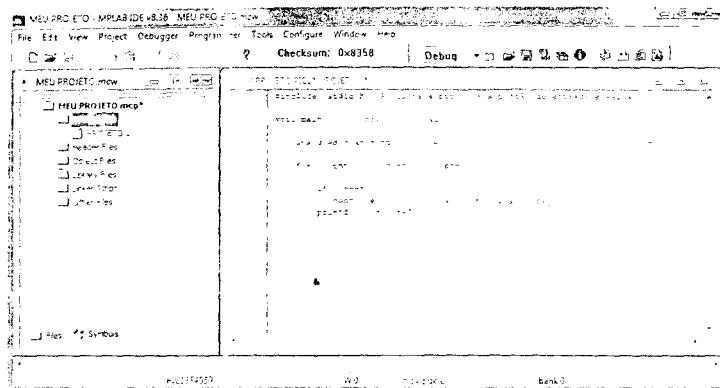


Figura 2.5

3 Configuração do Compilador MPLAB® C18

As opções de configuração do compilador MPLAB® C18 (Capítulo 3) podem ser acessadas pelo caminho **Project → Build Options... → Project aka MPLAB C18**.

A configuração do compilador MPLAB® C18 é realizada pela linha de comando, cujas opções estão organizadas em três categorias: *General*, *Memory Model* e *Optimization*.

A categoria *General* contém as opções de níveis de diagnóstico, classes de armazenamento, macros, entre outras opções básicas, Figura 2.6.

A categoria *Memory Model* contém as opções de memória relacionadas ao modelo do código, dado e pilha (stack), Figura 2.7.

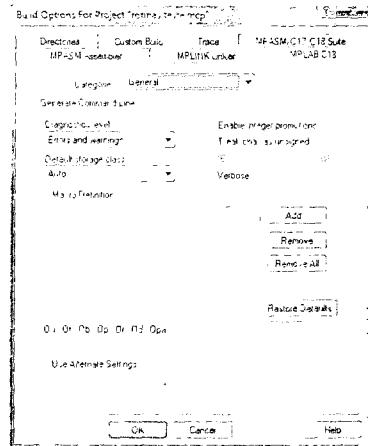


Figura 2.6

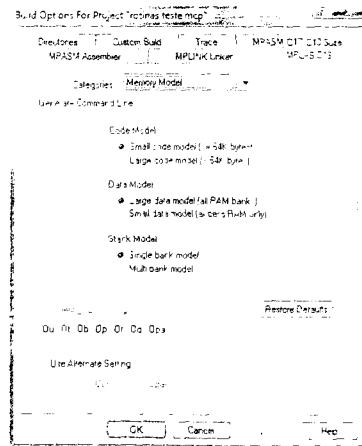


Figura 2.7

A última categoria é a *Optimization*, a qual comprehende as opções de otimização do código. Dentro dessa categoria é possível habilitar/desabilitar todas as funções de otimização, habilitar somente as otimizações suportadas pelo modo de debugação, além de possibilitar que o usuário habilite/desabilite algumas opções de otimização de acordo com o seu interesse.

Note que existe um campo chamado *Use Alternate Settings* comum a todas as categorias, que, quando selecionado, possibilita a introdução manual das opções da linha de comando, que serão explicadas detalhadamente no Capítulo 3.

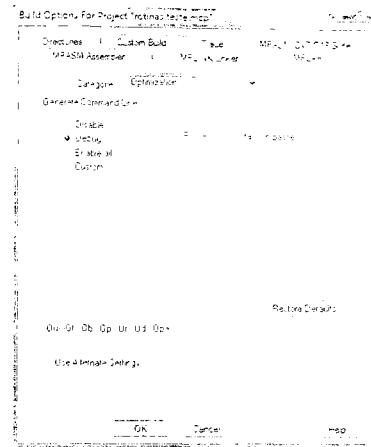


Figura 2.8

2.4 Configurações do Microcontrolador

As configurações do microcontrolador PIC® estão localizadas dentro da opção **Configure**.

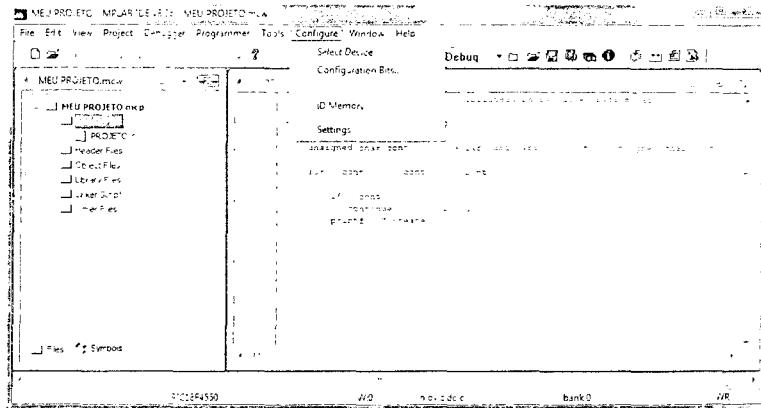


Figura 2.9

2.4.1 Select Device...

A opção **Select Device** permite que o usuário selecione o modelo do microcontrolador PIC®, o qual será utilizado no programa. Após selecionar o dispositivo, o programa informa quais são as ferramentas da Microchip que suportam o modelo selecionado.

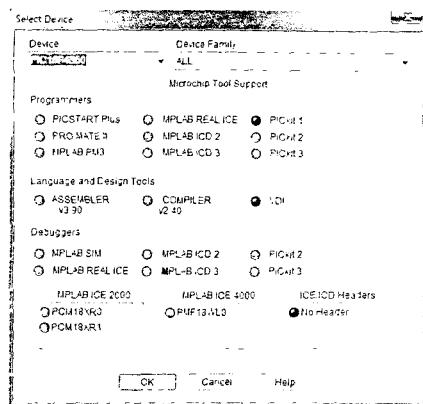


Figura 2.10

2.4.2 Configuration Bits...

A opção **Configuration Bits** abre uma tela que informa as configurações do dispositivo. Por meio dessa tela é possível visualizar os *bits* de configuração selecionados pela linha de código (`#pragma config`). Se o usuário preferir, também é possível selecionar os *bits* de configuração por meio dessa tela, desmarcando a caixa *Configuration Bits set in code*.

Configuration Bits				
Configuration Bits Set in Code				
Address	Value	Category	Setting	
300000	00	32MHz PIC Processor	No Clocks (4MHz I/O port)	
		CGU System Clock Postscaler	(OSC1/OSC2 Div: 2) [32MHz PIC Src: /4]	
300001	03	Full-Speed G3B Clock Source Selection	Clock source from OSC1/OSC2	
		PLL oscillator	EC_EG-CLOCK(R4), USB-EC	
300002	1F	Fail-Safe Clock Monitor Enable	Disabled	
		Internal External Switch Over Mode	Disabled	
300003	1F	Power Up Timer	Disabled	
		Brown Out Detect	Enabled in hardware, SREGEN disabled	
		Brown Out Voltage	2.3V	
300005	23	USB Voltage Regulator	Disabled	
		Watchdog Timer	Enabled	
		Watchdog Postscaler	1:32768	
300006	83	CCP2 Mux	RC1	
		PortB A/D Enable	PCARD<4> configured as analog inputs or RESET	
		Low Power Timer1 Osc enable	Disabled	
		Master Clear Enable	MCLR Enabled, R53 Disabled	
		Stack Overflow Reset	Enabled	

Figura 2.11

2.5 Compilação do Projeto

Clique em Project → Build All.

Se a compilação for bem-sucedida, aparece a mensagem *Build Succeeded* na última linha.

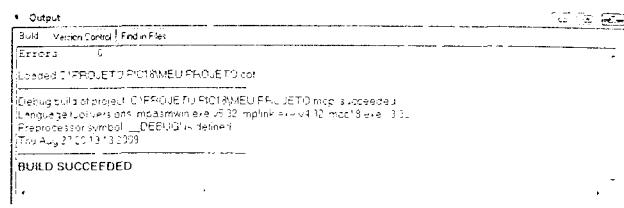


Figura 2.12

Caso ocorra algum erro durante a compilação, aparece uma tela informando o local onde se encontra o erro dentro do código-fonte, bem como sua descrição.

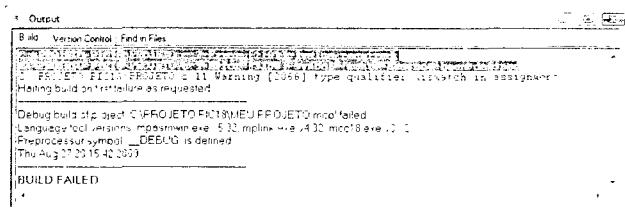


Figura 2.13



Para localizar o erro, clique duas vezes na linha.

2.6 Verificação da Quantidade de Memória de Dados e de Programa Utilizada pelo Código

Clique em View → Memory Usage Gauge.

Através desta tela, podemos analisar a quantidade total de memória de programa (*Program Memory*) e memória de dados (*Data Memory*) que o modelo de microcontrolador usado no projeto possui. Podemos também verificar a quantidade de memória ocupada pelo código (região hachurada), sendo que a memória de dados está expressa em *bytes* e a de programa, em *word*.

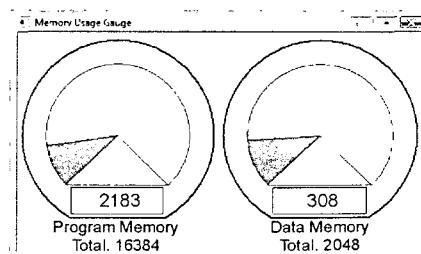


Figura 2.14

2.7 Visualização e Alteração do Conteúdo da Memória EEPROM Interna

Clique em View → EEPROM.

Para modificar os valores contidos na EEPROM, clique duas vezes no campo desejado e adicione o conteúdo.

Figura 2.15

asc queira armazenar caracteres, utilize o campo ASCII. Note que os pontos correspondem às colunas da
tabela em questão. Para acessar a coluna desejada, clique três vezes na coluna, então substitua o caractere.

1.8 Verificação e Alteração do Conteúdo dos Registradores : Variáveis do Projeto

Close em View → Watch.

Para adicionar registradores, selecione o registrador desejado pelo campo à direita do botão *Add SFR* e em seguida clique em *Add SFR*.

Watch			
Add Step	DATA	Add Symbol	DATA
Update	Address	Symbol Name	Value
	005	FCRA	3x3
			0x33333333

Figura 2.16

Para adicionar variáveis ou constantes utilizadas no programa, selecione pelo campo à direita do botão *Add Symbol* e em sequida clique em *Add Symbol*.



Note que os valores dos registradores/variáveis estão representados em binário. Caso o programador queira outro tipo de representação, clique com o botão direito do mouse sobre o registrador, variável ou constante desejada e selecione *properties...*.

2.9 Visualização das Variáveis Locais

Um modo simples e eficiente de verificar o conteúdo das variáveis locais é selecionar a opção Locals, clicando em View → Locals. Quando a janela está ativa, o programa lista todas as variáveis locais presentes na função que está sendo compilada.

Locals		
Address	Symbol Name	Value
31F	MAIN\$0	MAIN\$0
31E		00
31D		00
31C		00
31B		00
31A		00
319		00
318		00
317		00
316		00
315		00
314		00
313		00
312		00
311		00
310		00
309		00
308		00
307		00
306		00
305		00
304		00
303		00
302		00
301		00
300		00

Figura 2.17

2.10 Simulação e Depuração do Código-Fonte do Projeto

Clique em Debugger → Select Tool → <Selecione o depurador desejado>.

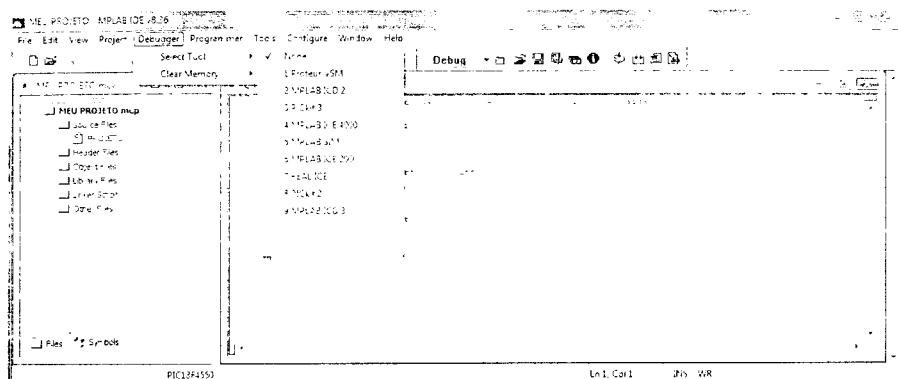


Figura 2.18

Exemplo

O exemplo desta seção utiliza o depurador MPLAB® SIM, pois ele não exige que o programador tenha em mãos um *hardware* dedicado.

Primeiramente selecione o depurador desejado, clicando em Debugger → Select Tool → MPLAB SIM. Note que aparece uma nova barra referente aos comandos de depuração.

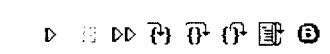


Figura 2.19

Sendo:

- ▶ : executa o código continuamente até encontrar um ponto de parada.
- ⏷ : interrompe a execução do código.
- ⏸ : executa o código pausadamente até encontrar um ponto de parada.
- ⏴ ⏵ ⏹ : executa o código passo a passo.
- ⏴ : reinicia a depuração.
- ⏷ : cria pontos de parada.



Para criar pontos de parada, clique duas vezes na linha desejada.

Em seguida clique em **Debugger** → **Stimulus** → **New Workbook**.

Após criar um Workbook, aparece a tela a seguir, Figura 2.20.

Na aba ASYNCH o programador pode simular valores nos pinos e registradores, além de definir o tipo de ação.

Existem seis abas, e cada uma delas permite que o programador simule os valores dos pinos/registradores de maneiras diferentes, de acordo com a necessidade de cada processo.

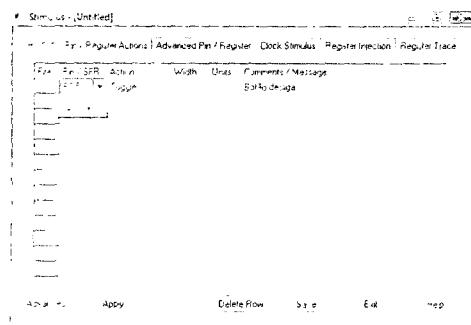


Figura 2.20

2.11 Simulação de UART pelo MPLAB® SIM

Um modo de testar as funções de envio de caracteres (`putc`, `puts`, `printf` etc.) é habilitar a **UART** do simulador MPLAB® SIM. Basta clicar em **Debugger** → **Setting** e abrir a aba **UART1 IO**, selecionar a caixa **Enable UART1 IO** e no campo **Output** selecionar o modo **Windows**. Após realizar este procedimento, sempre que for chamada alguma função de envio de caracteres com saída na **UART**, os dados serão impressos na janela **Output** na aba **SIM UART1**.

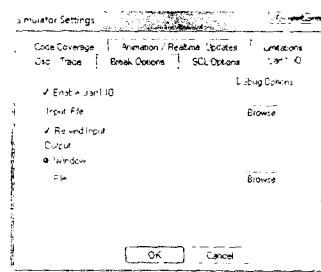


Figura 2.21

2.12 Gravação do Programa no Microcontrolador

Quando o programa já estiver rodando adequadamente, o programador pode gravar o código no microcontrolador com o auxílio de diferentes modelos de gravadores de microcontroladores PIC®. Neste exemplo utilizamos o modelo ICD 2.

Primeiramente conecte o gravador no computador e clique em **Programmer** → **Select Programmer** → **MPLAB ICD 2**.

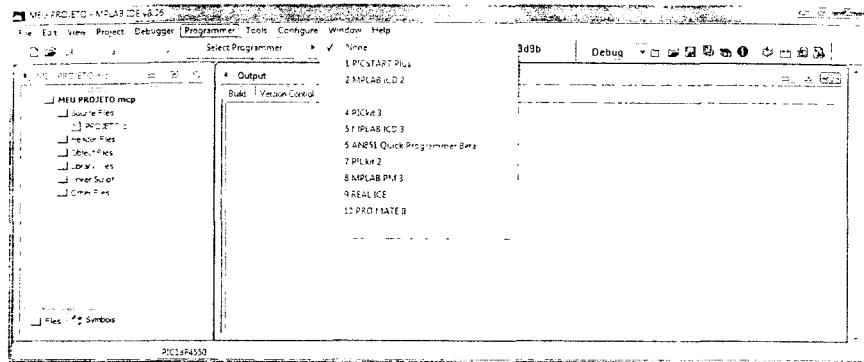


Figura 2.22

Clique em **Programmer** → **Connect**. Se o gravador for reconhecido, surge a tela apresentada em seguida:

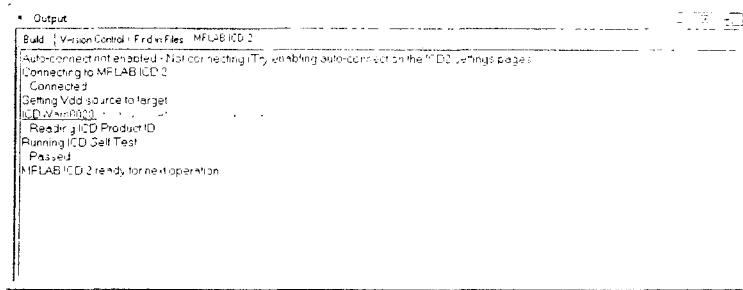


Figura 2.23

Caso ocorra um erro, siga os passos:

- Desconecte o gravador do computador.
- Clique em **Programmer** → **Select Programmer** → **NONE**.
- Conecte o gravador no computador.
- Clique em **Programmer** → **Select Programmer** → **MPLAB ICD 2**.

Se o gravador for iniciado corretamente, surge uma nova barra referente aos comandos de programação.



Figura 2.24

Sendo:

-  : grava o programa no microcontrolador.
-  : efetua a leitura do programa contido no microcontrolador.
-  : efetua a leitura do conteúdo da EEPROM do microcontrolador.
-  : verifica a memória do microcontrolador.
-  : apaga o conteúdo do microcontrolador.
-  : verifica se a memória do microcontrolador está vazia.
-  : reinicia e conecta o gravador.

Uma vez que o gravador está devidamente conectado, plugue o microcontrolador no gravador.

O microcontrolador pode ser programado de duas formas distintas. Uma delas o dispositivo deve ser retirado do circuito eletrônico ao qual pertence e deve ser posto em um suporte especial para gravação, e a outra forma é a gravação **in-circuit**, que é a mais eficiente, uma vez que não há necessidade de retirar o microcontrolador do circuito eletrônico para realizar a gravação do chip.

Após conectar o microcontrolador referenciado no projeto, efetue a gravação.



A disposição dos pinos do gravador não é padronizada, por este motivo leia atentamente o manual do gravador antes de efetuar a gravação.

3

Compilador MPLAB® C18

O MPLAB® C18 é um compilador C destinado aos microcontroladores da família PIC18. Ele possui compatibilidade com o padrão ANSI (*American National Standards Institute*) e pode ser facilmente integrado ao MPLAB® IDE. Suas principais características são:

- É compatível com o padrão ANSI.
- Permite misturar códigos C e Assembly em um único projeto.
- Permite configurar níveis de otimização do código.
- Dispõe de uma vasta biblioteca (PWM, SPI, I²C, USART etc.).

A versão demo deste compilador pode ser obtida através do site oficial da Microchip Technology (www.microchip.com).

3.1 Considerações Iniciais

O compilador MPLAB® C18 necessita de pelo menos três arquivos para compilar o programa C. Os arquivos com extensão '_e' são destinados ao modo estendido, que será explicado mais adiante.

O primeiro arquivo é a biblioteca C padrão representada pelos arquivos **plib.lib** ou **plib_e.iib**, o segundo é a biblioteca do processador (**p18xxxx.lib** ou **p18xxxx_e.lib**). Ambas bibliotecas estão localizadas no subdiretório **C:\MCC18\lib**.

O terceiro arquivo contém o código de inicialização do programa, chamado de arquivo objeto. Existem três versões de código de inicialização no MPLAB® C18, sendo **c018*.o** o destinado ao compilador que opera no modo não estendido e **c018*_e.o** para o modo estendido.

- **c018.o/c018_e.o:** inicializa a pilha e pula para o início da função principal, **main ()**.
- **c018i.o/c018i_e.o:** realiza as mesmas tarefas do arquivo anterior, além de designar os valores apropriados para as variáveis inicializadas antes de chamar a função **main ()**.
- **c018iz.o/c018iz_e.o:** desempenha as mesmas funções dos arquivos supracitados e também atribui o valor zero (0) às variáveis não inicializadas.

Os arquivos objeto podem ser encontrados no subdiretório **C:\MCC18\lib**.

3.2 Linha de Comando

O compilador MPLAB® C18 pode ser chamado pelo programa MPLAB® IDE (Capítulo 2) ou pelo *prompt* de comando do Windows OS. A linha de comando possui a sintaxe apresentada em seguida.

mcc18 <opção> <arquivo> <opção>

A linha de comando suporta apenas um arquivo-fonte (.c) e múltiplas opções, que serão apresentadas no decorrer desta seção. Com a linha de comando é possível controlar as funções do compilador, desde as mais básicas, como, por exemplo, o modo como as variáveis devem ser tratadas caso não sejam especificadas as classes de armazenamento, até as mais avançadas, como opções de otimização do código. Veja a figura apresentada em seguida:

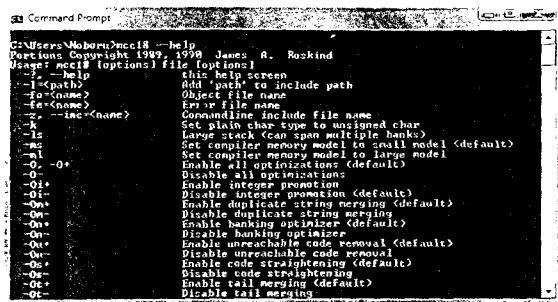


Figura 3.1: Linha de comando executada. `mcc18 --help`.

A opção `--help` lista todas as opções suportadas pela linha de comando e a opção `-verbose` mostra o número da versão do compilador e o número de erros, warnings e mensagens do arquivo compilado. Veja o exemplo em seguida.

C:\PROJETO PIC18> `mcc18 PROJETO.C -verbose`.

```

MPLAB C18 v3.32 academic Copyright 2000-2009 Microchip Technology Inc.
WARNING The procedural abstraction optimization is not supported in the academic version
Errors: 0
Warnings: 0
C:\PROJETO PIC18\PROJETO.C:9:Warning [2106] comparison of a signed integer to an unsigned integer detected
C:\PROJETO PIC18\PROJETO.C:12:Warning [2106] comparison of a signed integer to an unsigned integer detected
C:\PROJETO PIC18\PROJETO.C:14:Warning [2066] type qualifier mismatch in assignment
Errors: 0
Warnings: 3

```

Segue a lista das principais opções suportadas pela linha de comando.

Tabela 3.1: Lista das principais opções suportadas pela linha de comando.

Opção	Descrição
<code>-?, --help</code>	Lista todas as opções suportadas pelo compilador. Exemplo: <code>mcc18 --help</code>
<code>-I=<caminho></code>	Especifica o diretório onde se encontra o arquivo de cabeçalho adicionado pela diretiva <code>#include <arquivo></code> . Se o compilador não encontrar o <i>arquivo</i> no caminho especificado, ele passa a procurá-lo nos diretórios padrão do MPLAB® C18. Exemplo: <code>mcc18 -p=18f452 -I=C:\MCC18\h C:\pic18\teste.c</code>
<code>-fo=<nome></code>	Nome do arquivo objeto gerado através da compilação de um determinado arquivo-fonte. Exemplo: <code>mcc18 -I=C:\MCC18\h C:\pic18\teste.c -fo=c:\pic18\teste.o</code>
<code>-fe=<nome></code>	Nome do arquivo de erro gerado através da compilação de um determinado arquivo-fonte. Exemplo: <code>mcc18 -I=C:\MCC18\h C:\pic18\teste.c -fo=c:\pic18\teste.err</code>
<code>-k</code>	O compilador trata o tipo <code>char</code> como sendo do tipo <code>unsigned char</code> .

Opção	Descrição
-ls	Large Stack. Se uma pilha maior que 256bytes for usada, então deve-se ativar esta opção.
-ms	Define o modelo da memória do compilador para o <i>small code model</i> ($\leq 64\text{Kbytes}$) (padrão).
-ml	Define o modelo da memória do compilador para o <i>large code model</i> ($> 64\text{Kbytes}$).
-Oa+	Habilita como padrão, dado armazenado na memória de acesso.
	Observação: Válido somente para o modo não estendido.
-Oa-	Desabilita como padrão, dado armazenado na memória de acesso (padrão).
	Observação: Válido somente para o modo não estendido.
-sca	Classe de armazenamento padrão adotado pelo compilador: auto (padrão).
	Observação: Válido somente para o modo não estendido.
-scs	Classe de armazenamento padrão adotado pelo compilador: static .
	Observação: Válido somente para o modo não estendido.
-sco	Classe de armazenamento padrão adotado pelo compilador: overlay .
	Observação: Válido somente para o modo não estendido.
-pa=<n>	Define o número de repetições do <i>procedural abstraction</i> . Por definição n = 4.
	Observação: Veja as opções -Opa+/-Opa- .
-p=<processador>	Define o processador. O padrão adotado é genérico.
-D<macro>[=texto]	Define uma macro.
-w=<n>	Define níveis de aviso. O padrão adotado é 2. 1 - Erros (fatais e não fatais). 2 - Erros (fatais e não fatais) e warnings. 3 - Erros (fatais e não fatais), warnings e mensagens.
-nw=<n>	Suprime as mensagens especificadas por n. Exemplo: <code>mcc18 c:\pic18\untitled.c -nw=2056</code>
-v	Mostra a versão do compilador MPLAB® C18.
-verbose	Mostra a versão do compilador MPLAB® C18 e lista os erros e warnings do arquivo compilado. Exemplo: <code>mcc18 c:\pic18\teste.c -verbose</code>
--extended	Gera o código com instruções estendidas. Exemplo: <code>mcc18 c:\pic18\untitled.c --extended</code>
--no-extended	Gera o código sem instruções estendidas. Exemplo: <code>mcc18 c:\pic18\untitled.c --no-extended</code>
--help-message-list	Lista todas as mensagens de diagnóstico. Exemplo: <code>mcc18 --help-message-list</code>
--help-message-all	Mostra ajuda para todas as mensagens de diagnóstico. Exemplo: <code>mcc18 --help-message-all</code>
--help-message=<n>	Exibe ajuda para o diagnóstico de número n. Exemplo: <code>mcc18 --help-message=2104</code>
--help-config	Exibe ajuda sobre as definições de configuração de um determinado dispositivo. Exemplo: <code>mcc18 -p=18f4550 --help-config</code>

3.2.1 Opções de Otimização

O compilador MPLAB® C18, como a maioria dos compiladores, agrega funções de otimização do código, justamente para prover um código mais reduzido e eficiente. As opções de otimização do MPLAB® C18 estão listadas na Tabela 3.2.

Tabela 3.2: Lista das opções de otimização suportadas pela linha de comando

Opção	Descrição
-O, -O+	Habilita todas as otimizações. (padrão)
-O-	Desabilita todas as otimizações.
-Od+	Habilita a opção <i>dead code removal</i> . (padrão)
-Od-	Desabilita a opção <i>dead code removal</i> .
-Oi+	Habilita a opção <i>integer promotions</i> .
-Oi-	Desabilita a opção <i>integer promotions</i> . (padrão)
-Om+	Habilita a opção <i>duplicate string merging</i> . (padrão)
-Om-	Desabilita a opção <i>duplicate string merging</i> .
-On+	Habilita a opção <i>banking</i> . (padrão)
-On-	Desabilita a opção <i>banking</i> .
-Op+	Habilita a opção <i>copy propagation</i> . (padrão)
-Op-	Desabilita a opção <i>copy propagation</i> .
-Or+	Habilita a opção <i>redundant store removal</i> . (padrão)
-Or-	Desabilita a opção <i>redundant store removal</i> .
-Ou+	Habilita a opção <i>unreachable code removal</i> . (padrão)
-Ou-	Desabilita a opção <i>unreachable code removal</i> .
-Os+	Habilita a opção <i>code straightening</i> . (padrão)
-Os-	Desabilita a opção <i>code straightening</i> .
-Ot+	Habilita a opção <i>tail merging</i> (padrão)
-Ot-	Desabilita a opção <i>tail merging</i> .
-Ob+	Habilita a opção <i>branch</i> (padrão)
-Ob-	Desabilita a opção <i>branch</i> .
-Ow+	Habilita a opção WREG Content Tracking . (padrão)
-Ow-	Desabilita a opção WREG Content Tracking .
-Opa+	Habilita a opção <i>procedural abstraction</i> . (padrão)
-Opa-	Desabilita a opção <i>procedural abstraction</i> .

3.2.1.1 Dead Code Removal

O compilador considera código morto os valores calculados dentro de uma determinada função, os quais não são utilizados após a saída da função. O mesmo vale para instruções que computam somente valores mortos. Veja o exemplo em seguida.

Exemplo

```
int resp;
void f_2 (int var_1)
{
    int var_2;
    var_2 = var_1;
    resp = var_2;
}
```

A variável `var_2` recebe o valor de `var_1` e passa-o para a variável `resp`. Esse tipo de operação pode ser resumido em `resp = var_1`. Sendo assim, a `var_2` não tem utilidade nenhuma e pode ser eliminada do bloco de código. Veja como ficará esse código após otimizá-lo com as opções `dead code removal (-Od+)` e `copy propagation (-Op+)` habilitadas.

```
int resp;
void f_2 (int var_1)
{
    resp = var_1;
}
```

3.2.1.2 Integer Promotion

A ISO descreve que toda operação aritmética deve ser realizada com precisão do tipo `int` ou superior. Porém, o compilador MPLAB® C18, por definição, realiza operações aritméticas com o tamanho do maior operando, mesmo que este seja menor que o tipo `int`. Podemos adequar o MPLAB® C18 à norma ISO habilitando a opção `integer promotions (-Oit+)`. Veja no exemplo seguinte como essa opção afeta as operações aritméticas.

Exemplo

```
unsigned char x, y;
int z;
int resposta_1, resposta_2;
```

```
x = y = z = 128;
```

```
resposta_1 = x + y;
resposta_2 = x + z;
```

Tabela 3.3 Valores retornados pelas operações aritméticas sem/com a opção `integer promotion`.

<code>-Oi-</code>	<code>-Oit+</code>
<code>resposta_1 = 0</code> - prevalece o tipo <code>unsigned char</code> . <code>resposta_2 = 256</code> - prevalece o tipo <code>int</code> .	<code>resposta_1 = 256</code> - prevalece o tipo <code>int</code> . <code>resposta_2 = 256</code> - prevalece o tipo <code>int</code> .

3.2.1.3 Duplicate String Merging

Quando a opção `duplicate string merging` está habilitada (`-Om+`), o compilador pega duas ou mais `strings` literais idênticas e combina-as dentro de uma única tabela de `string` com uma única instância dos dados armazenados na memória de programa, ou seja, se declararmos duas `strings` literais idênticas (`const rom char *s1 = "TESTE";` e `const rom char *s2 = "TESTE";`), a `string` "TESTE" será armazenada em uma região da memória de programa e os ponteiros `s1` e `s2` vão apontar para a mesma região da memória de programa em que a `string` está localizada.

3.2.1.4 Banking

A otimização do banco remove a instrução `MOVLB`, caso seja verificado que o registro de seleção do banco (`BSR`) já possui o valor correto.

3.2.1.5 Copy Propagation

Copy propagation é um processo de otimização, em que o compilador substitui as ocorrências de uso da atribuição direta ($x = y$) com o valor atribuído, desde que as instruções intermediárias não modifiquem os valores de x e y .

Exemplo

```
int resp;
void f_2 (int var_1)
{
    int var_2, var_3 = 4;
    var_2 = var_1;
    resp = var_3 + var_2;
}
```

A variável `var_2` recebe o valor de `var_1`, soma o valor recebido com a `var_3` e armazena o resultado da operação em `resp`. Esse tipo de operação pode ser resumido em `resp = var_3 + var_1`. Veja como ficará este código após otimizá-lo com a opção *copy propagation* (`-Op+`) habilitada:

```
int resp;
void f_2 (int var_1)
{
    int var_2, var_3 = 4;
    var_2 = var_1;
    resp = var_3 + var_1;
}
```

Verifica-se que a variável temporária `var_2` é um código morto e pode ser eliminada se a opção *dead code removal* (`-Od+`) for habilitada.

```
int resp;
void f_2 (int var_1)
{
    int var_3 = 4;
    resp = var_3 + var_1;
}
```

3.2.1.6 Redundant Store Removal

Essa opção remove atribuições feitas várias vezes em uma sequência de instruções, desde que as instruções intermediárias não tenham alterado os valores das variáveis.

Exemplo

```
int resp;
void f_2 (int var_1)
{
    int var_2, var_3;

    resp = var_1;
    var_2 = 3;
    var_3 = resp + var_2
    resp = var_1;
}
```

Observe a ocorrência de duas atribuições diretas (`resp = var_1`) e que os valores de `var_1` e `resp` não são alterados pelas instruções intermediárias. Logo, se a opção *redundant store removal* (`-Or+`) estiver habilitada, o código gerado pelo compilador será:

```
int resp;
void f_2 (int var_1)
{
    int var_2, var_3;

    resp = var_1;
    var_2 = 3;
    var_3 = resp + var_2
}
```

3.2.1.7 Unreachable Code Removal

Essa opção elimina os códigos que não são executados pelo programa.

3.2.1.8 Tail Merging

A opção *tail merging* se aplica em blocos onde as últimas sequências de instruções são idênticas e continuam na mesma localização. Esse tipo de otimização substitui as instruções correspondentes de um bloco por um desvio para o ponto correspondente no outro bloco.

Exemplo

```
if ( valor )
    PORTC = 0x84;
else
    PORTC = 0x23;
```

Tabela 3.4. Código gerado com/sem a opção *tail merging*.

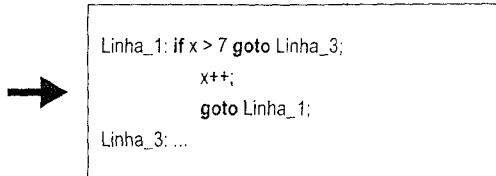
-Ot-	-Ot+
0x000000 MOVF valor, 0x0, 0x0	0x000000 MOVF valor, 0x0, 0x0
0x000002 BZ 0xa	0x000002 BZ 0x8
0x000004 MOVLW 0x84	0x000004 MOVLW 0x84
0x000006 MOVWF PORTC,0x0	0x000006 BRA 0xa
0x000008 BRA 0xe	0x000008 MOVLW 0x23
0x00000a MOVLW 0x23	0x00000a MOVWF PORTC,0x0
0x00000c MOVWF PORTC,0x0	0x00000c RETURN 0x0
0x00000e RETURN 0x0	

3.2.1.9 Branches

Quando essa opção está habilitada (`-Ob+`), as seguintes otimizações de desvio estão disponíveis:

- Um desvio (condicional ou incondicional) para um desvio incondicional pode ser modificado para dirigir-se ao último ponto em vez deste.

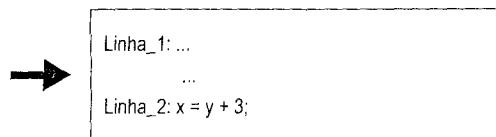
```
Linha_1: if x > 7 goto Linha_2;
        x++;
        goto Linha_1;
Linha_2: goto Linha_3;
Linha_3: .
```



```
Linha_1: if x > 7 goto Linha_3;
        x++;
        goto Linha_1;
Linha_3: ...
```

- Um desvio incondicional para uma instrução RETURN, ADDULNK ou SUBULNK pode ser substituído por uma instrução RETURN, ADDULNK ou SUBULNK respectivamente.
- Um desvio (condicional ou incondicional) para a instrução imediatamente seguinte ao desvio pode ser removido.

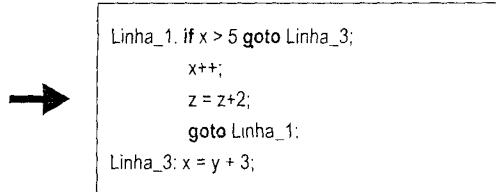
```
Linha_1: ...
...
goto Linha_2;
Linha_2: x = y + 3;
```



```
Linha_1: ...
...
Linha_2: x = y + 3;
```

- Um desvio condicional para um desvio condicional pode ser modificado para dirigir-se ao último ponto, se ambos os desvios estão sobre a mesma condição.

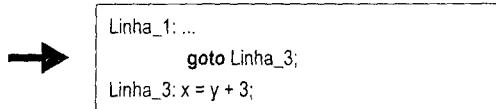
```
Linha_1 if x > 5 goto Linha_2;
        x++;
Linha_2: if x > y goto Linha_3;
        z = z+2;
        goto Linha_1;
Linha_3: x = y + 3;
```



```
Linha_1. if x > 5 goto Linha_3;
        x++;
        z = z+2;
        goto Linha_1;
Linha_3: x = y + 3;
```

- Um desvio condicional imediatamente seguido por um desvio incondicional para o mesmo destino pode ser removido (o desvio incondicional é suficiente).

```
Linha_1. if x > 5 goto Linha_3;
        goto Linha_3;
Linha_3: x = y + 3;
```



```
Linha_1: ...
        goto Linha_3;
Linha_3: x = y + 3;
```

3.2.1.10 Code Straightening

Reorganiza a sequência de códigos de forma que apareçam na ordem em que são executados.

3.2.1.11 Wreg Content Tracking

Quando o monitoramento do acumulador (**WREG**) está habilitado (**-Ow+**), o compilador suprime a instrução **MOVLW** se for verificado que o acumulador já possui o correto valor. Veja o exemplo seguinte.

Exemplo

```
unsigned char var_1, var_2;
var_1 = 0x34;
var_2 = 0x34;
```

Tabela 3.5: Códigos de saída sem/com monitoramento do acumulador.

-Ow-	-Ow+
0x000000 MOVLW 0x34	0x000000 MOVLW 0x34
0x000002 MOVWF var_1,0x1	0x000002 MOVWF var_1,0x1
0x000004 MOVLW 0x34	0x000004 MOVLW 0x34
0x000006 MOVWF var_2,0x1	0x000006 MOVWF var_2,0x1

3.2.1.12 Procedural Abstraction

O MPLAB® C18 geralmente cria um procedimento contendo o código repetido e o substitui por uma chamada de procedimento (-Opa+). Esse tipo de otimização reduz o tamanho do código, pois os códigos repetidos são substituídos por chamadas de procedimento (-Opa+) em vez de gerar todos os códigos (-Opa-).

Nem todos os códigos são capazes de serem abstraídos em uma única passagem do processo de abstração, o qual é realizado até não ocorrer mais abstrações ou até atingir o valor de repetições igual a 4 (padrão). As repetições (passos) podem ser configuradas através da opção `-pa=<n>`, e cada passo (n) adiciona 2ⁿ-1 níveis de chamadas de função. Como cada chamada de função ocupa um local na Stack (pilha), a opção `-pa=<n>` deve ser usada cuidadosamente em aplicações em que a Stack é limitada.

3.3 Modo Estendido (Extended Mode)

Alguns cuidados devem ser tomados quando o compilador está configurado no modo estendido. Por exemplo, o compilador suporta somente a classe **auto** como padrão e não dá suporte às classes **static** e **overlay** (seções de **overlay** são suportadas), gerando uma **warning** quando essas classes são encontradas no programa, além disso, não é possível definir como padrão a estocagem de dados na Access RAM.

O processador genérico (-p18cxx) selecionado nesse modo é o PIC18F4620, enquanto no modo não estendido é o PIC18C452.

Nesse modo o compilador passa a reconhecer oito instruções adicionais, as quais não são reconhecidas no modo não estendido, gerando um erro, caso seja verificada a ocorrência destas.

As instruções *Assembly* adicionadas no modo estendido são **ADDFSR**, **ADDULNK**, **CALLW**, **MOVSF**, **MOVSS**, **PUSHL**, **SUBFSR** e **SUBULNK**.



Linguagem C MPLAB® C18

Este capítulo aborda alguns conceitos básicos relacionados à linguagem de programação C. É importante ler atentamente, pois o compilador MPLAB® C18 possui algumas funções especiais que não fazem parte da linguagem C padrão.

4.1 Comentários

Inserir comentários no código de programa permite melhor organização e compreensão de seu funcionamento, já que é possível adicionará-lo a qualquer parte do programa. Ele desempenha um papel fundamental, pois permite que as instruções sejam comentadas uma a uma.

Há duas maneiras de introduzir comentários no programa.

Sintaxe

/* <comentário> - comentário de uma única linha.
 * <comentário> */ - comentário de múltiplas linhas.

Exemplo

```
/*
 Este código foi desenvolvido somente para exemplificar os comentários.
 */
#include "C:\MCC18\h\p18f4550.h" //Arquivo de cabeçalho do PIC18F4550.
void main(void) //Função principal.
{ /*Rotina... */ }
```

4.2 Identificadores

O nome dado às variáveis, funções, rótulos ou outros elementos da linguagem C é conhecido como identificador. Em um programa não é permitido identificadores com o mesmo nome. Isso significa que, se há uma variável cujo nome é **resultado**, não é permitido que uma função também tenha o nome de **resultado**. Todo identificador começa com uma letra ou um caractere **_**, e o restante dos caracteres que constituem o nome deve ser número, letra ou **_**. Outro ponto importante é que somente os 31 primeiros caracteres são relevantes. Veja na Tabela 4.1 alguns exemplos de identificadores.

Tabela 4.1: Identificadores válidos x inválidos.

Identificador válido	Identificador inválido
Cont	Cont-32
Valor_canal	1Valor_canal

Identificador válido	Identificador inválido
_teste	@teste
teste22	Teste...22

4.3 Palavras-Chaves

As palavras-chaves são conjuntos de palavras reservadas pela linguagem e não podem ser utilizadas como identificadores. Veja a tabela seguinte:

Tabela 4.2: Palavras-chaves

auto	break	case	char	const	continue	default	do
double	else	enum	extern	far	float	for	goto
if	int	long	near	overlay	ram	register	return
rom	short	signed	sizeof	static	struct	switch	typedef
union	unsigned	void	volatile	while			

4.4 Tipos de Dados

Os tipos de dados suportados pelo compilador MPLAB® C18 estão listados na Tabela 4.3.

Tabela 4.3: Tipos de dados.

Tipo	Valor	Tamanho
char	-128 a 127	8bits
unsigned char	0 a 255	8bits
signed char	-128 a 127	8bits
int	-32.768 a 32.767	16bits
unsigned int	0 a 65.535	16bits
short	-32.768 a 32.767	16bits
unsigned short	0 a 65.535	16bits
short long	-8.388.608 a 8.388.607	24bits
unsigned short long	0 a 16.777.215	24bits
long	-2.147.483.648 a 2.147.483.647	32bits
unsigned long	0 a 4294967295	32bits
float	1,17549435e - 38 a 6.80564693e + 38	32bits
double	1,17549435e - 38 a 6.80564693e + 38	32bits

Um aspecto importante que deve ser considerado refere-se ao tamanho do dado a ser inserido em um determinado tipo de variável. Caso o valor inserido seja superior à capacidade do tipo da variável, o valor real armazenado na variável em questão será equivalente ao resto da divisão entre o valor inserido e o tamanho da variável. Veja o exemplo apresentado em seguida:

```
unsigned char var_teste; //Cria uma variável de 8bits chamada var_teste.  
var_teste = 300; //Atribui um valor superior à capacidade da variável.
```

Var_teste é uma variável de 8bits, logo temos que $2^8 = 256$, significando que a variável do tipo **unsigned char** é capaz de armazenar valores entre 0 e 255, Tabela 4.3, o que equivale a um tamanho de 256. No entanto, o valor inserido é 300. Como ele é maior que a capacidade da variável var_teste, o valor real presente nessa variável é igual a 44, o que significa que a instrução var_teste=300 é o mesmo que var_teste = 44. O valor 44 pode ser obtido pelo seguinte cálculo:

$$\frac{\text{Valor_inserido}}{2^{\text{Tamanho_da_variável_em_bits}}} = \frac{300}{2^8} = \frac{300}{256} = 1\frac{44}{256}$$

4.5 Tipos de Qualificadores

Os qualificadores, também conhecidos como modificadores, podem ser divididos em qualificadores de tipo de acesso e de armazenamento.

4.5.1 Qualificadores de Armazenamento

Eles informam ao compilador o modo como os dados devem ser acessados ou modificados. O compilador MPLAB® C18 suporta os qualificadores padrão **volatile** e **const**, e introduz mais quatro qualificadores denominados **far**, **near**, **ram** e **rom**.

- **volatile**: indica que o conteúdo da variável pode ser modificado a qualquer momento durante a execução do programa.
- **const**: mostra que o valor da variável não pode ser alterado pelo programa. Sempre que o qualificador **const** for utilizado, é necessário atribuir um valor no momento da declaração da variável.

Agora que já vimos os dois qualificadores padrão, vamos comentar sobre os qualificadores introduzidos pelo compilador MPLAB® C18.

Sabemos que a arquitetura do PIC® é do tipo *Harvard*, por este motivo o MPLAB® C18 precisa de uma extensão para distinguir se o dado está localizado na RAM ou na ROM, pois não compartilham o mesmo barramento. Essa distinção é feita através da introdução dos qualificadores **ram** e **rom** antes dos qualificadores **far** e **near**. Um objeto definido sem um qualificador é por definição **far** e **ram**. Agora veremos como esses qualificadores atuam na memória de dados e de programa.

4.5.1.1 Memória de Dados

- **ram far**: indica ao compilador que a variável deve ser alocada em um banco de memória de dados (GPR), e uma instrução de seleção de banco é necessária antes de acessar essa variável.
- **ram near**: informa ao compilador que a variável deve ser alocada no Access RAM.

Exemplo

`ram far int var_1; //Variável localizada na atual seção da memória de dados (GPR).`

```
#pragma udata access meu_dado_nao_iniciado=0x00
    ram near long var_2; //Variável localizada no endereço 0x00 da Access RAM.
#pragma udata
```

4.5.1.2 Memória de Programa

- **rom far**: indica ao compilador que a variável pode ser armazenada em qualquer lugar da memória de programa. Se for um ponteiro, pode acessar além dos 64K do espaço da memória de programa.
- **rom near**: informa ao compilador que a variável deve ser armazenada na memória de programa, porém o endereço não pode ultrapassar 64K. Se for um ponteiro, pode acessar somente até 64K do espaço da memória de programa.

Exemplo

```
rom far unsigned int var_1=4938; //Variável localizada em qualquer endereço da memória de programa.
rom near long var_2=228394; //Variável localizada em um endereço da memória de programa menor que 64K.
```

```
#pragma romdata overlay minha_rom=0x400
    rom char mem[ ]={"012345"} //Armazena a string na memória de programa [0x0400 - 0x0406]
#pragma romdata
```

4.5.1.3 Ponteiros

Ponteiros podem apontar para a memória de dados (**ram**) ou de programa (**rom**), cujo tamanho é definido pela tabela seguinte:

Tabela 4.4: Tipos de ponteiros.

Tipo de ponteiro	Exemplo	Tamanho
Ponteiro de memória de dados	char * dmp;	16bits
Ponteiro de memória de programa (near)	rom near char * npmp;	16bits
Ponteiro de memória de programa (far)	rom far char * fpmp,	24bits

4.5.2 Classes de Armazenamento

As classes de armazenamento informam ao compilador o modo como os dados devem ser armazenados. Entre elas temos: **auto**, **static**, **extern**, **register** e **overlay**.

- **auto**: mostra que a variável existirá enquanto o procedimento estiver ativo. Esse especificador é pouco usado, pois é adotado como padrão.
- **static**: essa classe funciona de modo diferente para variáveis locais ou globais. Variáveis globais declaradas com **static** são conhecidas somente dentro do módulo (arquivo) em que foram criadas, não podendo ser acessadas por outros módulos. Quando esse tipo de classe é utilizado em variáveis locais, essas variáveis conservam o valor mesmo que o programa esteja executando outra função. Isso ocorre porque as variáveis possuem um armazenamento global, porém são conhecidas somente dentro da função na qual foram criadas. Veja o exemplo em seguida.

```
#include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída.
```

```
int teste_static( )
{
    static int contador=0;
    return contador++;
}
void main( void )
{
    while(1) { fprintf ( stdout, "\nChamadas da função teste_static --> %u", teste_static( ) ); }
}
```

Executando o código passo a passo, verifica-se a seguinte mensagem de saída:

Mensagem impressa

```
Chamadas da função teste_static --> 1
Chamadas da função teste_static --> 2
Chamadas da função teste_static --> 3
```

Se a variável *contador* não fosse declarada com a classe **static**, ela seria inicializada sempre que a função *teste_static()* fosse chamada e seu valor sempre seria igual a 1 (um).



A classe **static** é válida somente para o modo não estendido.

- **extern:** informa que a variável global já foi declarada em outro arquivo (módulo), porém será utilizada no arquivo corrente. Desta forma o valor é simplesmente alocado.

Para melhor compreendermos esse tipo de classe, suponha que dois arquivos (*Arquivo1.c* e *Arquivo2.c*) serão unidos, porém apresentam uma variável em comum (*resultado_mult*).

Arquivo1.c

```
long resultado_mult;
int x, y;
```

```
void main( void )
{
    x=40, y=32;
    resultado_mult = multiplicacao(x,y);
}
```

Arquivo2.c

```
extern long resultado_mult;

long multiplicacao(int a, int b)
{
    resultado_mult = a*b;
    return resultado_mult;
}
```

Se a declaração da variável *resultado_mult* localizada no *Arquivo2.c* não fosse feita com a classe **extern**, o compilador geraria um erro informando a ocorrência de multiplicidade de declaração.

- **register:** indica ao compilador que a variável declarada deve ser alocada no registrador da CPU. Nos microcontroladores PIC®, esse tipo de classe não faz sentido, uma vez que as variáveis já são armazenadas nos registradores da CPU (GPR - General Purpose Register).
- **overlay:** essa classe pertence exclusivamente ao compilador MPLAB® C18 e sua função é alocar estaticamente variáveis locais e parâmetros. Sempre que possível, as variáveis locais serão sobrepostas com variáveis locais de outra função. A classe **overlay** é válida somente para o modo não estendido.

4.6 Instrução

Uma instrução é uma expressão seguida de um ';'. Muitas instruções podem estar dentro de { e } para formar uma instrução composta ou bloco que tem uma sintaxe equivalente a uma instrução.

Exemplo

```
val_1 = a/b; //Instrução 1.
val_2 = a*c; //Instrução 2.
```

4.7 Declaração

Declaração é uma instrução composta de um tipo de especificador (tipo de dado) e de uma lista de identificadores (nome das variáveis ou constantes) separados por uma vírgula.

Na linguagem C as variáveis do programa podem ser declaradas como locais ou globais. As variáveis locais são declaradas dentro de uma determinada função, as quais serão válidas somente para essa função, enquanto as variáveis globais são válidas para todas as funções, e declaradas no início do programa. Veja o exemplo em seguida.

```
char v_global; //Variável global.

char funcao_1() //Função secundária.
{
    int v_local_1; //Variável local
    ...
}

int funcao_2() //Função secundária
{
    int v_local_2; //Variável local
    ...
}

void main () //Função primária
{
    int v_local_m; //Variável local
}
```

Exemplo:

```
int a; //Declaração de uma variável de 16bits
long b = 9;c; //Declaração de duas variáveis de 32bits com atribuição de valor à variável b.
float y = 2.38e4; //Declaração de uma variável de 32bits com atribuição de valor à variável y.
signed int val_1, val_2; //Declaração de duas variáveis de 16bits com sinal.
```



Toda variável deve ser declarada antes de ser utilizada.

4.8 Representações dos Dados Numéricos

Os dados numéricos são representados na base binária, octal, hexadecimal e decimal. Essa flexibilidade da linguagem C possibilita ao programador manipular o valor das variáveis em quatro formatos distintos. Veja na Tabela 4.5 a representação das bases.

Tabela 4.5: Representação dos dados numéricos.

Representação do valor	Base numérica
0b01010111	Binária
065	Octal
0x57	Hexadecimal
87	Decimal

4.9 Matrizes

Matriz é um agrupamento de variáveis do mesmo tipo, associadas a um nome. Ela pode ser unidimensional (uma única dimensão) ou multidimensional (mais de uma dimensão) e cada posição da matriz corresponde a uma variável do tipo especificado, sendo acessada por um índice.

4.9.1 Unidimensional

Matriz unidimensional armazena os elementos em apenas uma dimensão. A declaração desse tipo de matriz pode ser vista em seguida.

Sintaxe

tipo identificador [quantidade_posições].

Sendo:

- **tipo:** tipo de dado válido.
- **identificador:** nome da matriz.
- **quantidade_posições:** número de elementos da matriz.

Matrizes unidimensionais podem ser representadas conforme a Figura 4.1.

Posição	n	.	3	2	1	0
Conteúdo						
Tamanho da matriz						

Figura 4.1: Matriz unidimensional.



A primeira posição da matriz é sempre zero. Isso implica que o índice de uma matriz de cinco elementos varia de 0 a 4, totalizando cinco posições.

Exemplo

```
void main( ) // Função principal.
{
// Declara uma matriz de cinco elementos do tipo unsigned char com atribuição de valores.
// Posição: 0 1 2 3 4
    unsigned char erro[5] = {12,33,45,66,70};
    unsigned char numero_erro; // Declara uma variável do tipo unsigned char para armazenar o valor do erro.
    numero_erro = erro[3]; // A variável numero_erro recebe o valor (66) presente na posição especificada.

    erro[0] = 192; // Sobrescreve o valor da posição 0 da matriz.
    numero_erro = erro[0]; // A variável numero_erro recebe o valor (192) presente na posição especificada.
}
```

A matriz declarada no exemplo tem atribuição de valores às posições, porém não é obrigatório. Caso não sejam atribuídos valores iniciais à matriz, os valores presentes nas posições dela serão lixo de memória, por este motivo, todos os campos devem ser preenchidos adequadamente durante o programa, para que não ocorram inconsistências nos dados.

4.9.2 Multidimensional

Matriz multidimensional armazena os elementos em mais de uma dimensão. A declaração desse tipo de matriz pode ser vista em seguida.

Sintaxe

tipo identificador [quantidade_pos_x] [quantidade_pos_y] . . [quantidade_pos_z].

Sendo:

- **tipo:** tipo de dado válido.
- **identificador:** nome da matriz.
- **quantidade_pos:** número de elementos da matriz.

Para facilitar a compreensão de matrizes multidimensionais pegamos como exemplo uma matriz linha x coluna (bidimensional).

A Figura 4.2 apresenta uma matriz 3x4 do tipo **char** (declaração: **char mat[3][4]**).

		C	0	1	2	3
L	0	'a'	'b'	'v'	'k'	
	1	'g'	'h'	't'	'n'	
	2	'e'	'o'	'b'	'm'	

Figura 4.2: Matriz bidimensional.



A primeira posição da matriz é sempre zero. Isso implica que o índice de uma matriz de três linhas e quatro colunas varia de 0 a 2 e 0 a 3 respectivamente.

Exemplo

```
char mat[3][4] = { 'a', 'b', 'v', 'k', 'g', 'h', 't', 'n', 'e', 'o', 'b', 'm' }; // Declara uma matriz 3x4 do tipo char
char caractere; // Declara uma variável do tipo char para armazenar o caractere da matriz
caractere = mat[1][3]; // A variável caractere recebe o caractere ('n') presente na posição especificada
```

4.10 Operadores

A linguagem C dispõe de uma grande gama de operadores, se comparada com outras linguagens. Os operadores apresentados nesta seção são de atribuição, aritméticos, bit a bit, lógicos, relacionais e ponteiros.

4.10.1 Atribuição

Os operadores de atribuição são utilizados sempre que houver a necessidade de passar um dado a uma determinada variável.

Sintaxe

identificador = expressão;

Sendo:

- **identificador:** nome da variável.
- **expressão:** pode ser uma constante, operação matemática, função etc.

Exemplo

```
int x, y, z = 30; // Declara três variáveis do tipo int (16bits) e atribui o valor 30 decimal à variável z
long res_1; // Declara uma variável do tipo long (32bits).
```

```
x = 3; // Atribui o valor 3 decimal à variável x.
y = 5; // Atribui o valor 5 decimal à variável y.
```

```
res_1 = x + y + z; // A variável res_1 armazena o resultado da operação matemática.
```

4.10.2 Aritméticos

Os operadores aritméticos realizam operações matemáticas entre constantes e/ou variáveis do mesmo tipo ou de tipos diferentes.

Tabela 4.6. Operadores aritméticos.

Operador	Ação
+	Adição
+=	Adição
++	Incremento de 1
-	Subtração
-=	Subtração
--	Decremento de 1
*	Multiplicação
*=	Multiplicação
/	Divisão
/=	Divisão
%	Retorna o resto inteiro da divisão
%=	Retorna o resto inteiro da divisão

Exemplo

```
unsigned char x, y, z; // Declara três variáveis do tipo unsigned char (8bits).
float res; // Declara uma variável do tipo float (32bits).
```

```
x = 36; // Atribui o valor 36 decimal à variável x.
y = 10; // Atribui o valor 10 decimal à variável y.
z = 25; // Atribui o valor 25 decimal à variável z.
```

```
x += 19; //x = 36 + 19 = 55
res = (x + z)/y; //res = (55 + 25)/10 = 8
res = x % y; //res = 55 % 10 = 5
```

4.10.3 Bit a Bit

Operadores bit a bit manipulam bits de variáveis inteiras do tipo **int**, **long**, **long long** e **char**. Com eles é possível realizar operações **and**, **or**, **xor**, **not**, além de deslocar bits para direita ou esquerda.

Tabela 4.7: Operadores bit a bit.

Operador	Ação
&	AND (E)
	OR (Ou)
^	XOR (Ou exclusivo)
~	NOT (inverte o estado dos bits)
>>	Deslocamento de bit(s) à direita
<<	Deslocamento de bit(s) à esquerda

Exemplo

```
unsigned char x, y, z; // Declara três variáveis do tipo unsigned char (8bits)
unsigned int res; // Declara uma variável do tipo unsigned int (16bits).
```

```
x = 0b00101001; // Atribui o valor binário 0010 1001 à variável x.
y = 0b10001000; // Atribui o valor binário 1000 1000 à variável y.
z = 0b01110101; // Atribui o valor binário 0111 0101 à variável z.
```

```
res = x & y; // res = 0000 0000 0000 1000
res = y | z; // res = 0000 0000 1111 1101
res = ((res&0x00ff)&y)<<8; // res = 1000 1000 0000 0000
res |= x; // res = 1000 1000 0010 1001
res = ~res; // res = 0111 0111 1101 0110
```

4.10.4 Relacionais

Os operadores relacionais são normalmente utilizados em comandos de decisão para verificar se uma determinada condição é verdadeira (1) ou falsa (0).

Tabela 4.8: Operadores relacionais.

Operador	Ação
==	Igual a
!=	Diferente de
>	Maior que
>=	Maior ou igual a
<	Menor que
<=	Menor ou igual a

Veja alguns exemplos listados na Tabela 4.9.

Tabela 4.9: Valor retornado pelas expressões.

Condição	Resultado	Valor retornado
44 == 56	Falso	0
14 != 66	Verdadeiro	1
34 > 34	Falso	0
34 >= 34	Verdadeiro	1
56 < 100	Verdadeiro	1
87 <= 33	Falso	0

4.10.5 Lógicos

Os operadores lógicos são frequentemente utilizados em conjunto com operadores relacionais. Com eles é possível verificar se a combinação de uma ou mais condições resulta em uma condição verdadeira (1) ou falsa (0).

Tabela 4.10: Operadores lógicos booleanos.

Operador	Ação
&&	AND (E)
	OR (Ou)
!	NOT (Não)

Veja alguns exemplos listados na tabela seguinte.

Tabela 4.11: Valor retornado pelas expressões.

Condição	Descrição	Resultado	Valor retornado
$(44 == 56) \parallel (14 != 66)$	$0 \parallel 1 = 1$	Verdadeiro	1
$(44 == 56) \&\& (14 != 66)$	$0 \&\& 1 = 0$	Falso	0
$!(31 > 34) \&\& (34 >= 34)$	$!0 \&\& 1 = 1$	Verdadeiro	1
$((34 >= 34)) \parallel ((44 == 56)) \&\& !(14 != 66)$	$(1 \parallel 0) \&\& 1 = 1$	Verdadeiro	1
$((34 >= 34) \&\& ((44 == 56)) \&\& !(14 != 66))$	$(1 \&\& 0) \&\& 1 = 0$	Falso	0
$((34 >= 34) \&\& (44 == 56)) \parallel (14 != 66)$	$(1 \&\& 0) \parallel 1 = 1$	Verdadeiro	1

4.10.6 Ponteiros

Ponteiro é uma variável que contém um endereço de memória. Ele é normalmente utilizado para simular a passagem de parâmetros por referência, possibilitando que uma função retorne mais de um valor, além de referenciar elementos de matrizes.

Tabela 4.12 Operadores de ponteiros.

Operador	Ação
&	Endereço do operando.
*	Conteúdo do endereço apontado pelo operando.

A declaração de um ponteiro tem o seguinte formato.

Sintaxe

tipo * *identificador*;

Sendo:

- **tipo**: tipo de dado válido.
- **identificador**: nome da variável ou ponteiro.

Exemplo

int mat[4] = { 4, 13, 67, 70 }; // Declara uma matriz de quatro elementos do tipo int, com atribuição de valores.

int *p; // Declara um ponteiro para uma variável do tipo int (16bits).

int elemento_0, elemento_1, elemento_2, elemento_3; // Declara quatro variáveis do tipo int.

```
p = &mat[0]; // O ponteiro p armazena o endereço do primeiro elemento da matriz mat.
elemento_0 = *p; // O ponteiro p retorna o conteúdo do endereço apontado - elemento_0 = 4
p++, // Incrementa o endereço em um
elemento_1 = *p; // O ponteiro p retorna o conteúdo do endereço apontado - elemento_1 = 13
p++; // Incrementa o endereço em um
elemento_2 = *p; // O ponteiro p retorna o conteúdo do endereço apontado - elemento_2 = 67
p++; // Incrementa o endereço em um
elemento_3 = *p; // O ponteiro p retorna o conteúdo do endereço apontado - elemento_3 = 70
```

4.11 Funções

As funções são indispensáveis na linguagem C, pois elas contêm blocos de comandos que definem as ações do programa. A forma geral de uma função pode ser representada de acordo com a seguinte sintaxe.

Sintaxe

```
tipo_de_dado_retornado nome_da_função ( parâmetros )
{
    bloco_de_comandos
    return valor_retornado;
}
```

Sendo:

- **tipo_de_dado_retornado**: tipo de valor devolvido pelo comando **return**.
- **valor_retornado**: é o valor devolvido pela função.
- **bloco_de_comandos**: comandos executados pela função.
- **nome_da_função**: nome da função.
- **parâmetros**: é uma lista de variáveis declaradas, separadas por vírgula. Essas variáveis armazenam os valores dos argumentos passados durante a chamada da função. e serão utilizadas pela função. (Opcional)
- **{ e }**: indicam o início e o fim da função.

As funções podem ser classificadas em principal e secundária. Dentro de um programa em C, podem existir diversas funções secundárias, no entanto é permitido somente uma função principal.

A função principal tem o seguinte formato.

Sintaxe

```
void main ( void )
{ bloco_de_comandos }
```

Sendo:

- **bloco_de_comandos**: comandos executados pela função.
- **{ e }**: indicam o início e o fim da função.

A função principal contém a rotina central do programa. Com ela é possível chamar as funções secundárias, que podem chamar outra função secundária. Porém, uma função secundária não pode chamar uma função primária.

As funções secundárias são criadas sempre que uma determinada rotina se repete ou executa uma operação especial. Elas possibilitam maior clareza do código de programa, além de poupar memória de programa, uma vez que substituem rotinas repetidas.

Exemplo

```

int compara(int a, int b) // Função secundária com passagem de argumentos.
{
    if (a>b) //Verifica se x é maior que y.
        return 0; // Sai da função e retorna 0. x é maior que y.
    else
    {
        if(a<b) //Verifica se x é menor que y.
            return 1; // Sai da função e retorna 1. x é menor que y.
        else //Se x não for maior nem menor que y, então x é igual a y.
            return 2; // Sai da função e retorna 2. x é igual a y.
    }
}

void main( void ) //Função principal.
{
    int res, x, y; // Declara três variáveis do tipo int

    res = compara ( 30, 15 ); // Chama a função compara () e armazena o valor retornado. (res = 0)
    res = compara ( 44, 44 ); // Chama a função compara () e armazena o valor retornado. (res = 2)
    x = 12; y = 19;
    res = compara ( x, y ); // Chama a função compara () e armazena o valor retornado. (res = 1)
}

```

A linguagem C só permite o acesso aos códigos e dados de uma função, se ela for chamada por outra função. As variáveis declaradas dentro de uma função (variáveis locais) existem somente para a função na qual foram declaradas. Veja o exemplo em seguida.

Exemplo

```

void soma( ) // Função secundária sem passagem de parâmetros.
{
    res = x + y; // Ocorre um erro, pois as variáveis res, x e y não existem para a função soma ( ).
}

void main( ) //Função principal.
{
    int res, x = 19, y = 16; // Declara três variáveis do tipo int com atribuição de valores às variáveis x e y.
    soma ( ); // Chama a função soma ( );
}

```

Se as variáveis `res`, `x` e `y` forem declaradas como globais, então nenhum erro será gerado, pois elas existirão para qualquer função do programa.

Exemplo

```

int res, x = 19, y = 16; // Declara três variáveis do tipo int com atribuição de valores às variáveis x e y

void soma( ) // Função secundária sem passagem de parâmetros
{
    res = x + y; } // Efetua a soma de x e y. (res = 35)

void main( ) //Função principal.
{ soma ( ); } /* Chama a função soma ( ); */

```

Outro ponto importante é que toda função deve ser declarada antes de ser chamada por outra função, ou seja, se pegarmos o exemplo anterior e invertermos as posições das funções, o compilador irá indicar um erro, pois a função `soma` foi declarada depois da função `main`.



Uma função não pode ser declarada dentro de outra função.

4.12 Comandos de Seleção

Os comandos de seleção são empregados quando se deseja executar um determinado tipo de ação com base no resultado de uma expressão condicional. Os comandos de seleção existentes na linguagem C são **if** e **switch**.

4.12.1 Comando If

O comando **if** executa um ou mais comandos se a **expressão** for verdadeira (diferente de '0'); caso contrário, executa o bloco de comandos presente no **else**. A cláusula **else** é opcional.

Sintaxe

```
if( expressão )
{ bloco_de_comandos }
else
{ bloco_de_comandos }
```

Sendo:

- **bloco_de_comandos**: comandos executados.
- **expressão**: expressões condicionais.
- **{ e }**: indicam o inicio e o fim do comando **if** e **else**.

Exemplo

```
void main( ) //Função principal.
{
    int cont = 6; // Declara uma variável do tipo int.
    short cond; // Declara uma variável do tipo short.

    if ( cont <= 5 ) //Verifica se a expressão é verdadeira.
    { cond = 1; } //Indica condição verdadeira.
    else //Se a expressão for falsa, a instrução dentro da cláusula else é executada.
    { cond = 0; } //Indica condição falsa.
}
```



Também é possível inserir uma condição **if** dentro da cláusula **else**.

4.12.2 Comando Switch

O comando **switch** verifica em uma lista de **constants** se o valor da **expressão** é verdadeiro. Se for igual ao rótulo (**constante_1**, **constante_2**, ..., **constante_n**) de um **case**, as instruções pertencentes a ele serão executadas. Se porventura nenhum **case** atender à condição, então os comandos presentes no **default** serão executados. O **default** é opcional.

Sintaxe

```

switch ( expressão )
{
    case constante_1 :
        bloco_de_comandos
        break.
    case constante_2 :
        bloco_de_comandos
        break.
    case constante_n :
        bloco_de_comandos
        break.
    default :
        bloco_de_comandos
}

```

Sendo:

- **bloco_de_comandos**: comandos executados.
- **constante**: constante testada. Pode ser uma constante inteira ou caractere.
- **expressão**: dado a ser testado pelo comando.
- { e }: indicam o inicio e o fim do comando **switch**.

Em todos os **case** existe o comando **break** que é indispensável, pois efetua a saída imediata do comando **switch**. Se não existir em algum dos **case**, as instruções do próximo **case** serão executadas até que um comando **break** seja encontrado ou o último **case** seja executado.

Exemplo

```

unsigned char int_para_char(int numero) // Função secundária
{
    unsigned char caractere_pro; // Declara uma variável do tipo unsigned char.

    switch (numero)
    {
        case 0: caractere_pro = '0'; break;
        case 1: caractere_pro = '1'; break;
        case 2: caractere_pro = '2'; break;
        case 3: caractere_pro = '3'; break;
        default: caractere_pro = '?'; break;
    }
    return caractere_pro; // Sai da função e retorna o valor do caractere_pro.
}

void main( ) //Função principal.
{
    unsigned char num; // Declara uma variável do tipo unsigned char.
    unsigned char caractere; // Declara uma variável do tipo unsigned char.

    num = 2; //Atribui o valor 2 decimal à variável num.
    caractere = int_para_char(num); // Chama a função int_para_char( ) e armazena o caractere retornado.
}

```

4.13 Laços

Os laços são empregados quando se deseja executar uma ou mais instruções repetidas vezes, enquanto uma determinada condição for verdadeira. Os laços existentes na linguagem C são:

- while
- do-while
- for

4.13.1 Laço While

O **while** é normalmente empregado em situações em que o laço pode ser finalizado a qualquer momento, devido à ligação entre a expressão e as ações executadas dentro do laço.

Sintaxe

```
while ( expressão )
{ bloco_de_comandos }
```

Sendo:

- **bloco_de_comandos**: comandos executados.
- **expressão**: expressões condicionais.
- { e }: indicam o início e o fim do comando **while**.

Exemplo

```
#include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída.
```

```
void main( ) //Função principal
{
    unsigned char contador=0;

    while(contador<10)
    {
        printf("Valor do contador: %u\n", contador); //Imprime o valor do contador.
        contador++;
    }
}
```

As instruções contidas no laço **while** serão executadas enquanto o valor da variável **contador** for inferior a 10.

4.13.2 Laço Do-While

O laço **do-while** é muito parecido com o laço **while**. A diferença entre eles é que o laço **while** testa a condição e depois executa o corpo do laço, enquanto o laço **do-while** executa o corpo do laço e depois testa a condição.

Sintaxe

```
do
{ bloco_de_comandos } while ( expressão );
```

Sendo:

- **bloco_de_comandos**: comandos executados.
- **expressão**: expressões condicionais
- **{ e }**: indicam o início e o fim do comando **do-while**.

Exemplo

```
#include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída

void main( ) //Função principal
{
    unsigned char contador=0;
    do
    {
        printf("Valor do contador: %u\n", contador);
        contador++;
    }while(contador<10);
}
```

As instruções contidas no laço **do-while** serão executadas enquanto o valor da variável *contador* for inferior a 10.

4.13.3 Laço For

O laço **for** é geralmente utilizado quando se deseja repetir uma ou mais instruções uma quantidade de vezes predefinida.

Sintaxe

```
for ( inicialização ; expressão ; incremento_decremento )
{ bloco_de_comandos }
```

Sendo:

- **inicialização**: valor inicial para a variável de controle.
- **expressão**: expressões condicionais.
- **incremento_decremento**: incrementa ou decrementa a variável de controle a cada repetição do laço.
- **bloco_de_comandos**: comandos executados.
- **{ e }**: indicam o início e o fim do comando **for**.

Exemplo

```
int cont; //Declara uma variável do tipo int.
int v_num[6]; //Declara uma matriz de seis posições do tipo int.
```

```
for ( cont = 0 ; cont < 6 ; cont++ )
{ v_num[cont] = cont; }
```

O laço **for** é executado enquanto o valor da variável *cont* for menor que 6. Isso significa que ele vai repetir seis vezes, pois o valor atribuído à variável de controle é 0 e o incremento é de uma unidade. Os elementos do vetor *v_num* são completados com o valor de *cont* a cada repetição do laço.

4.14 Comandos de Desvio

A linguagem C dispõe de comandos que possibilitam o desvio incondicional do programa, ou seja, através desses comandos é possível sair de um laço ou de um comando de seleção a qualquer momento, ignorando as expressões condicionais. Os comandos de desvio são **break**, **continue**, **goto** e **return**.

4.14.1 Comando Break

O comando **break** é utilizado para finalizar a execução de um laço (**for**, **while**, **do-while**) ou de um comando **switch**.

Exemplo

```
int cont = 0; //Declara uma variável do tipo int com atribuição de valor
while (1) //Looping infinito
{
    if (cont == 4)
        { break; } // Força a saída do laço while.
    cont++;
}
```

O laço **while** é executado enquanto o valor da variável *cont* for diferente de 4, pois em seu corpo de programa existe um comando **if**, cuja expressão é verdadeira quando o valor da variável *cont* é igual a 4. Então, uma instrução **break** realiza a saída do laço.

4.14.2 Comando Continue

O comando **continue** é semelhante ao comando **break**. A diferença é que o comando **continue** força a próxima iteração de um laço (**for**, **while**, **do-while**) em vez de forçar a saída.

Sempre que esse comando é utilizado, as instruções abaixo dele são ignoradas e uma nova iteração é iniciada.

Exemplo

```
#include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída.

void main( ) //Função principal
{
    unsigned int cont = 0; //Declara uma variável do tipo unsigned char (8 bits) com atribuição de valor.

    for ( cont = 0 ; cont < 6 ; cont++ )
    {
        if ( cont == 4 )
            { continue; } //Pula para a próxima iteração.
        printf ( "\nteste" );
    }
}
```

O laço **for** envia a *string* teste pela *stream* de saída, exceto quando a variável *cont* for igual a 4 ou ≥ 6 . No momento em que a variável *cont* for igual a 4, a condição **if** será verdadeira e a instrução **continue** será executada, desta forma o controlador salta a função **printf()** e uma nova iteração é iniciada.

4.14.3 Comando Goto

O comando **goto** força o desvio do controle para um determinado ponto dentro da função, identificado pelo **rótulo**.

Sintaxe
goto rótulo ,
...
rótulo :
...

Sendo:

- **rótulo**: é um identificador acompanhado de ':'.

Exemplo

`int cont; //Declara uma variável do tipo int.`

```
for ( cont = 0 ; cont < 6 ; cont++ )
{
    if ( cont == 4 )
        { goto fim; } // Desvia o contador para o rótulo fim.
}
fim.
//Inserir rotina
```

Quando o laço **for** é executado, a variável *cont* passa a ser incrementada a cada iteração. Ao atingir um valor equivalente a 4, a condição **if** passa a ser verdadeira e o comando **goto** desvia o contador para a posição do rótulo indicado (*fim*), e o programa continua a ser executado a partir deste ponto.

4.14.4 Comando Return

O comando **return** retorna um valor de uma função. Ele pode ser encontrado em qualquer parte da função a qual foi chamada e sempre que for executado, a função é finalizada e um valor é retornado.

Sintaxe
return expressão ;

Sendo:

- **expressão**: é qualquer tipo de expressão que retorne um valor.

Esse comando pode ser empregado em qualquer tipo de função, exceto a **void**, pois funções desse tipo não retornam valores.

Exemplo

```
long operacao(int num_opcao, int a, int b) // Função secundária
{
    switch (num_opcao)
    {
        case 1: return ( a + b ); //Retorna o resultado da soma dos elementos a e b.
        case 2: return ( a - b ); //Retorna o resultado da subtração dos elementos a e b.
        case 3: return ( a * b ); //Retorna o resultado da multiplicação dos elementos a e b.
        case 4: return ( a / b ); //Retorna o resultado da divisão dos elementos a e b.
```

```

    default: return 0; //Retorna 0.
}
}

void main( ) //Função principal.
{
    unsigned char x = 50, y = 10; // Declara duas variáveis do tipo unsigned char com atribuição de valor.
    long resultado=0; // Declara uma variável do tipo long.

    //Suponha que as opções são. 1 – soma, 2 – subtração, 3 – multiplicação, 4 – divisão.
    resultado = operacao(1, x, y); // Chama a função operacao ( ) e armazena o valor retornado. (resultado = 60)
    resultado = operacao(2, x, y); // Chama a função operacao ( ) e armazena o valor retornado (resultado = 40)
    resultado = operacao(3, x, y); // Chama a função operacao ( ) e armazena o valor retornado (resultado = 500)
    resultado = operacao(4, x, y); // Chama a função operacao ( ) e armazena o valor retornado. (resultado = 5)
}

```

4.15 Enumerações, Estruturas, Tipos de Dados Definidos pelo Usuário e Uniões

A linguagem C possui quatro palavras-chaves que permitem criar tipos de dados especiais. A **enum** (enumeração) cria uma lista de símbolos com valores constantes, a **struct** (estrutura) permite agrupar diversas variáveis e associá-las a um nome em comum, a **typedef** (tipo de dado definido pelo usuário) designa outro nome a um tipo de dado válido e por último temos a **union** (união) que possibilita que diferentes tipos de variáveis compartilhem uma mesma posição da memória.

4.15.1 Enumerações

As enumerações (**enum**) definem um grupo de constantes inteiras. O primeiro símbolo (*constante_1*) assume o valor zero e os subsequentes têm um incremento de um. Por este motivo podemos relacionar valores aos símbolos. Outro ponto importante é a possibilidade de atribuir valores aos símbolos, porém os símbolos subsequentes a este terão o valor do símbolo anterior incrementado de um.

Sintaxe

```
enum identificador { constante_1, constante_2, ... , constante_n } variáveis_enumeração;
```

Sendo:

- **identificador**: nome da enumeração.
- **constante**: nome das constantes.
- **variáveis_enumeração**: variáveis desse tipo de enumeração. (Opcional)

Exemplo

```
int valor_cor;
enum cores { vermelho, verde = 6, azul } numero_cor, cor;
```

```
/* vermelho = 0 / verde = 6 / azul = 7 */
```

```
void main( ) // Função principal.
```

```
{
    numero_cor = azul;
    valor_cor = numero_cor; // valor_cor = 7;
}
```

4.15.2 Estruturas

Uma estrutura (**struct**) é um conjunto de variáveis relacionadas a um nome. Ela se diferencia de uma matriz, pois os diversos elementos da estrutura não compartilham necessariamente a mesma zona de memória.

Elas são empregadas quando se deseja agrupar informações de diversos tipos de variáveis (Sintaxe 1) ou armazenar diversas variáveis *booleanas* (0 ou 1) em um único *byte* (Sintaxe 2).

Sintaxe 1

```
struct identificador
{
    tipo  membro_1;
    tipo  membro_2;
    tipo  membro_n;
} variáveis_estrutura;
```

Sintaxe 2

```
struct identificador
{
    tipo  membro_1 : comprimento_bit;
    tipo  membro_2 : comprimento_bit;
    tipo  membro_n : comprimento_bit;
} variáveis_estrutura;
```

Sendo:

- **identificador**: nome da estrutura.
- **tipo**: tipo de dado válido.
- **membro**: elementos da estrutura.
- **comprimento_bit**: tamanho ocupado pelo *membro*, em *bits*.
- **variáveis_estrutura**: variáveis desse tipo de estrutura. (Opcional)

Exemplo

```
struct aluno // Nome da estrutura.
{
    int num_matricula; // Elemento da estrutura.
    float nota[4];   // Elemento da estrutura.
    float media;     // Elemento da estrutura.
} alberto; // Declara uma variável alberto do tipo aluno.

/* A estrutura pode ser definida de outro modo, como mostrado em seguida.
struct aluno // Nome da estrutura.
{
    int num_matricula; // Elemento da estrutura.
    float nota[4];   // Elemento da estrutura.
    float media;     // Elemento da estrutura.
};

struct aluno alberto; // Declara uma variável alberto do tipo aluno.
*/
main( ) // Função principal.
{
    alberto.num_matricula = 52;
    alberto.nota[0] = 60;
    alberto.nota[1] = 80;
```

```

alberto.nota[2] = 90;
alberto.nota[3] = 50;
alberto.media = (alberto.nota[0] + alberto.nota[1] + alberto.nota[2] + alberto.nota[3])/4; //alberto media = 70.
}

```

Exemplo

```

struct status// Nome da estrutura
{
    int ocupado:1;                      //Bit 0
    int reiniciar_apagar:1;              //Bit 1
    int comando_ilegal:1;                //Bit 2
    int crc:1;                           //Bit 3
    int seq_comando_apagar:1;            //Bit 4
    int endereco:1;                     //Bit 5
    int parametro:1;                    //Bit 6
    int irrelevante:1;                  //Bit 7
}cartao;// Declara uma variável cartao do tipo status

main( ) // Função principal.
{
    // Zera o status do cartão.
    cartao.ocupado = 0;
    cartao.reiniciar_apagar = 0;
    cartao.comando_ilegal = 0;
    cartao.crc = 0;
    cartao.seq_comando_apagar = 0;
    cartao.endereco = 0;
    cartao.parametro = 0;
    cartao.irrelevante = 0;

    // Escrever o resto da rotina.
}

```

4.15.3 Tipos de Dados Definidos pelo Usuário

A palavra-chave **typedef** define um novo nome a um determinado tipo de dado válido. Por exemplo, com essa palavra-chave é possível definir que o nome "caractere" (*novo_nome*) é sinônimo do tipo de dado **char** (*tipo*). Logo, ambos os nomes podem ser utilizados na declaração de variáveis.

Sintaxe

```
typedef tipo novo_nome;
```

Sendo:

- **tipo**: tipo de dado válido.
- **novo_nome**: novo nome para o tipo de dado representado pelo parâmetro *tipo*.

Exemplo

```
typedef char caractere; // Define um novo nome ao tipo de dado char
caractere letra; // Declara uma variável do tipo char denominada letra.
```

4.15.4 Uniões

A palavra-chave **union** compartilha uma posição da memória com diferentes tipos de variáveis, que serão utilizadas em momentos diversos. Desta forma, o espaço reservado na memória corresponde ao tamanho do maior membro da união, ou seja, se os membros da união forem do tipo **int** (16bits) e **long** (32bits), o tamanho da memória ocupada será de 32bits.

Sintaxe

```
union identificador
{
    tipo membro_1;
    tipo membro_2;
    tipo membro_n;
} variáveis_união;
```

Sendo:

- **identificador**: nome da união.
- **tipo**: tipo de dado válido.
- **membro**: elementos da união.
- **variáveis_união**: variáveis desse tipo de união. (Opcional)

Exemplo

union cont // Nome da união.

```
{
    int contador_16bits; // Elemento da união.
    long contador_32bits; // Elemento da união.
}contador; // Declara uma variável contador do tipo cont.

void main( ) // Função principal.
{
    contador.contador_16bits=0; //Atribui o valor zero ao contador_16bits.
    while (contador.contador_16bits<2339) //Looping
    {
        //Rotina.
        contador.contador_16bits++;
    }
    contador.contador_32bits=0; //Atribui o valor zero ao contador_32bits.

    while (contador.contador_32bits<2322379) //Looping
    {
        //Rotina.
        contador.contador_32bits++;
    }
}
```

A vantagem apresentada no exemplo anterior é a possibilidade de colocar contadores de diversos tamanhos em uma mesma posição de memória, resultando em economia na memória de dados (RAM).

4.16 Diretivas Básicas

As diretivas são comandos de pré-processador que não são compilados, e podem ser inseridas em qualquer parte do programa, porém não na mesma linha. Elas são iniciadas com o caractere **#** e não são terminadas com **'**. Com elas é possível incluir arquivos, definir macros, configurar o modo de operação do compilador etc.

4.16.1 #Define e #Undef

A diretiva `#define` define uma *string* que vai substituir o *identificador*, sempre que ele for utilizado no programa, enquanto o `#undef` elimina a definição de um determinado *identificador*.

Sintaxe

```
#define identificador string
#define identificador
```

Sendo:

- **identificador**: nome da constante.
- **string**: texto ou macros.

Podemos, por exemplo, utilizar identificadores definidos para fixar o valor máximo de iterações de um laço `for`.

Exemplo

```
#define MAX 4 //Define uma constante de nome MAX.
```

`void main() //Função principal.`

```
{
    int cont = 0, iteracao = 0; //Declara duas variáveis do tipo int com atribuição de valor.

    for ( cont = 0 ; cont < MAX ; cont++ )
        { iteracao++; } //Incrementa a variável iteração a cada repetição do laço for.
}
```

Outra aplicação dessa diretiva é a definição de macros.

Exemplo

```
#define INVERTE(var) (~var) //Define uma macro para INVERTE.
```

`void main() //Função principal.`

```
{
    unsigned int var = 0b0000110011011101; //Declara uma variável do tipo unsigned int com atribuição de valor.
    unsigned int var_invert; //Declara uma variável do tipo unsigned int.

    var_invert = INVERTE(var); //Inverte os bits da variável var. (11110011 00100010)
}
```

4.16.2 #Error

As diretivas `#error` forçam o compilador a gerar um código de erro no local onde se encontra a diretiva.

Sintaxe

```
#error mensagem
```

Sendo:

- **mensagem**: texto informando o erro.

Exemplo

1. `#include<p18f4550.h>` //Arquivo de cabeçalho do PIC18F4550.
- 2.
3. `#define TAMANHO_BUFFER 28` //Define uma constante de nome TAMANHO_BUFFER.
- 4.
5. `#if TAMANHO_BUFFER > 20` //Verifica se o TAMANHO_BUFFER é maior do que 20.

```

6.      #error Buffer super dimensionado //Gera um erro na janela output do compilador.
7.
8.      #elif TAMANHO_BUFFER < 10 //Verifica se o TAMANHO_BUFFER é menor do que 10.
9.
10.     #error Buffer insuficiente //Gera um erro na janela output do compilador.
11.
12.     #endif
13.
14.
15. void main( )
16. {
17.     char mensagem[TAMANHO_BUFFER]; //Declara uma matriz do tipo char.
18. }
```

Condições de erro: TAMANHO_BUFFER > 20 ou TAMANHO_BUFFER < 10.

Se o TAMANHO_BUFFER for definido como 28, teremos na janela de saída do compilador:

C:\Users\Noboru\Desktop\C18\Rotinas para\teste.c:8:Error [1099] Buffer superdimensionado

Caso o TAMANHO_BUFFER for definido como 9, teremos na janela de saída do compilador:

C:\Users\Noboru\Desktop\C18\Rotinas para\teste.c:12:Error [1099] Buffer insuficiente

4.16.3 #IF

A diretiva **#if** é um comando de pré-processador parecido com o comando **if**, pois ambos têm a função de verificar a validade de uma determinada expressão e executar uma ação. No entanto, a expressão contida em **#if** só é verificada durante a compilação, e isso significa que as expressões presentes nessa diretiva não suportam variáveis da linguagem C, somente constantes, operadores-padrão e identificadores criados pela diretiva **#define**.

Sintaxe

```

#if expressão
    códigos
#elif expressão
    códigos
#else
    códigos
#endif
```

Sendo:

- **expressão:** expressões condicionais.
- **códigos:** códigos de programa.



As diretivas **#elif** e **#else** são opcionais. Sempre que a diretiva **#if** for utilizada, ela deve ser finalizada através da diretiva **#endif**.

4.16.4 #ifdef e #ifndef

A diretiva **#ifdef** é uma forma abreviada da diretiva **#if defined(nome)**. Ela é utilizada para evitar a redefinição de operadores, constantes e identificadores, que podem eventualmente estar presentes nos arquivos inseridos no programa. A diretiva **#ifndef** é a forma negativa do **#ifdef**.

Sintaxe
`#ifdef nome`
 códigos
`#endif`

ou

`#ifndef nome`
 códigos
`#endif`

Sendo:

- **nome**: operadores, constantes ou identificadores
- **códigos**: códigos de programa.

Exemplo

```
#include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída

#define valor 930
#define valor_max 430

#if !defined(valor_min)
#define valor_min 10
#endif

#ifndef valor
#define valor 400
#endif

#endif valor_max
#define valor_max
#define valor_max 512
#endif

void main (void)
{
    printf(stdout, "nvalor=%u",valor);
    printf(stdout, "nvalor_min=%u",valor_min);
    printf(stdout, "nvalor_max=%u",valor_max);
}
```

Mensagem impressa

```
valor=930
valor_min=10
valor_max=512
```

4.16.5 #Include

Arquivos de cabeçalho e bibliotecas diversas podem ser adicionados ao programa através da diretiva `#include`. Há duas formas de incluir um arquivo externo. A primeira é inserir o nome do arquivo entre os símbolos '<' e '>' e a outra entre aspas duplas " ".

Sintaxe
`#include < arquivo >`
`#include " arquivo "`

Sendo:

- **arquivo:** nome do arquivo.

Quando o nome do arquivo se encontra entre os símbolos '<' e '>', o compilador procura o arquivo nos diretórios padrões do MPLAB® C18. Caso contrário, é necessário informar o local em que o arquivo se encontra.

4.16.6 #LINE

A diretiva **#line** modifica o conteúdo das macros **_LINE_** e **_FILE_**. Quando utilizada, o conteúdo fornecido pela macro **_FILE_** sempre corresponde à *string* definida pela diretiva **#line** e o valor devolvido pela macro **_LINE_** tem como referência o valor definido pelo parâmetro *valor_line* da diretiva **#line**.

Sintaxe

#line *valor_line* *string_file*

Sendo:

- **valor_line:** valor do tipo inteiro adotado pela macro **_LINE_**.
- **string_file:** string adotada pela macro **_FILE_**.

Exemplo

```
1. #include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
2. #include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída
3.
4. #line 54 "teste.c"
5.
6. void main (void)
7. {
8.     printf("Linha: %u \n", __LINE__);
9.     printf("Arquivo: " __FILE__ "\n");
10. }
```

Mensagem impressa

Linha: 58

Arquivo: teste.c

4.16.7 #Pragma

A diretiva **#pragma** instrui o compilador a executar uma ação específica durante o processo de compilação. Ela não possui funções-padrão e pode variar de acordo com o compilador. Por exemplo, no MPLAB® C18 é possível selecionar os *bits* de configuração do microcontrolador, definir a região da memória em que serão armazenados os dados ou blocos de código etc.

Existem diversas aplicações para a diretiva **#pragma** e algumas delas serão comentadas nos próximos capítulos, de modo a facilitar o aprendizado sobre as funcionalidades dessa diretiva.

4.16.7.1 #Pragma Sectiontype

A aplicação mais comum da diretiva **#pragma** no compilador MPLAB® C18 é identificar a seção da memória a ser utilizada no microcontrolador PIC18. Seções são as várias áreas da memória (dado ou programa) do PIC18, podendo conter código ou dados.

A diretiva **#pragma** dispõe de quatro maneiras para identificar a seção da memória a ser usada no PIC18, cujas palavras-chaves associadas são **code**, **romdata**, **udata** e **idata**

- **code:** informa ao compilador que o código logo abaixo da diretiva deve ser inserido na seção da memória de programa. Veremos mais adiante que é possível especificar a área da memória de programa onde o código será inserido, permitindo um total controle da localização do código na memória.
- **romdata:** indica ao compilador que as variáveis ou constantes declaradas logo abaixo da diretiva devem ser inseridas na seção da memória de programa. Esses dados são normalmente declarados com o qualificador **rom**.
- **udata:** informa ao compilador que as variáveis globais (alocadas estaticamente e **não** inicializadas) declaradas logo abaixo da diretiva devem ser armazenadas na seção da memória de dados.
- **idata:** mostra ao compilador que as variáveis globais (alocadas estaticamente e **inicializadas**) declaradas logo abaixo da diretiva devem ser armazenadas na seção da memória de dados. Diferentemente do modo **udata**, os dados declarados são inicializados, o que implica que o valor atribuído às variáveis está localizado em algum lugar da memória de programa e é movido para os registros de arquivo (dado) pelo código de inicialização do compilador antes de iniciar a execução.

Sintaxe

```
#pragma udata atributo_r identificador=endereço
#pragma idata atributo_r identificador=endereço
#pragma romdata atributo_p identificador=endereço
#pragma code atributo_p identificador=endereço
```

Sendo:

- **atributo_r:** pode ser atributo **access** ou **overlay**. (Opcional)
- **atributo_p:** é o atributo **overlay**. (Opcional)
- **identificador:** identificador.
- **endereço:** endereço da memória de programa ou de dado.

O atributo **access** informa ao compilador para localizar uma seção especificada em uma região da *Access RAM*, ou seja, o compilador deixa de usar o *BANK SELECT REGISTER* (BSR) e usa o *BANK ACCESS*. Variáveis localizadas nessa seção devem ser declaradas com a palavra-chave **near**.

```
#pragma udata access meu_dado_nao_iniciado=0x00
    near int var_1, var_2;
#pragma udata

#pragma idata access meu_dado_iniciado=0x10
    near unsigned char var_3='p', var_4=49;
#pragma idata
```

Se for utilizado o qualificador **near**, o objeto será localizado em uma memória acessível (*Access RAM*); caso contrário, se a definição de um objeto tiver o qualificador **far**, então o objeto será localizado em uma memória não acessível.

O atributo **overlay** permite que outras seções sejam localizadas no mesmo endereço físico e podem ser usadas em conjunto com o atributo **access**. Para sobrepor duas seções, quatro requisitos devem ser atendidos:

1. Cada seção deve residir em um diferente arquivo-fonte.
2. Ambas as seções devem possuir o mesmo nome.
3. Se o atributo **access** for especificado com uma seção, ele deve ser especificado com a outra.
4. Se um endereço absoluto for especificado com uma seção, o mesmo endereço deve ser especificado na outra.

O exemplo seguinte mostra a seção de código ocupando a mesma região da memória.

Arquivo1.c

```
#pragma code overlay meu_codigo=0x1300
void rotina_teste_1()
{ /*Rotina...*/ }
```

Arquivo2.c

```
#pragma code overlay meu_codigo =0x1300
void rotina_teste_2()
{ /*Rotina...*/ }
```

O exemplo seguinte mostra a seção de dado ocupando a mesma região da memória.

Arquivo1.c

```
#pragma udata overlay meu_dado=0x100
int var_1, var_2;
```

Arquivo2.c

```
#pragma udata overlay meu_dado=0x100
long var;
```

Vejamos agora como inserir códigos e constantes em regiões específicas da memória de programa sem o atributo **overlay**.

```
#pragma romdata meu_dado_rom=0x1000
rom unsigned char matriz_rom[]={0,'1','2','3','4','5','6','7','8','9'};

#pragma code meu_codigo=0x1300
void rotina_teste1( )
{ /*Rotina...*/ }

void rotina_teste2( )
{ /*Rotina...*/ }
```



Dúvidas com relação aos endereços da memória? Veja o mapa da memória de dados e de programa do PIC18F4550 no Capítulo 5.

4.16.7.2 #Pragma **Tmpdata**

Muda a seção em que o compilador deve armazenar os dados temporários. As declarações seguidas do *nome_seção* serão armazenadas nessa seção até que seja encontrada outra diretiva **#pragma tmpdata** sem especificar o nome da seção. Deste ponto em diante, por definição, o compilador MPLAB® C18 passa a armazenar os dados temporários na seção **.tmpdata**; caso contrário, se a diretiva apresentar o nome da seção (*nome_seção*), o compilador dá início ao armazenamento dos dados temporários nessa nova seção.

Sintaxe

#pragma tmpdata nome_seção

Sendo:

- **nome_seção**: nome da seção onde o compilador deverá armazenar os dados temporários.

4.15.7.3 #Pragma Varlocate

Informa ao compilador o **banco** em que as variáveis (*nome_variável*) estão explicitamente localizadas, resultando na remoção da instrução de seleção de banco (**MOVLB**) e, consequentemente, gerando um código mais eficiente.

Sintaxe

```
#pragma varlocate banco nome_variável
#pragma varlocate nome_seção nome_variável
```

Sendo:

- **banco**: banco da memória.
- **nome_variável**: nome de uma ou mais variáveis localizadas no banco, separadas por uma vírgula ','.
- **nome_seção**: nome da seção em que as variáveis estão localizadas.



Este comando simplesmente informa onde se encontram as variáveis, ele não faz a alocação.

Exemplo

Suponha que existam dois arquivos no projeto. Arquivo_1.c e Arquivo_2.c.

Arquivo_1.c

```
#pragma iodata banco_2=0x200 int var_1=4,
int var_2=12;
```

Arquivo_2.c

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.

#pragma varlocate 2 var_1, var_2 //Informa a localização das variáveis.
extern signed char var_1;
extern signed char var_2;

void main (void)
{ var_1 = 3; var_2 = 10; }
```

O código assembly apresentado em seguida representa o código C destacado no Arquivo_2.c.

```
MOVLB 0x2
MOVLW 0x3
MOVWF 0, BANKED
MOVLW 0xa
MOVWF 0x2, BANKED
```

Se o programa não possuísse a linha **#pragma varlocate 2 var_1, var_2**, o código gerado pelo compilador seria o apresentado em seguida.

```

MOVLB 0x2
MOVLW 0x3
MOVWF 0, BANKED
MOVLB 0x2
MOVLW 0xa
MOVWF 0x2, BANKED

```

4.17 Macros Predefinidas

O compilador MPLAB® C18 dispõe das macros predefinidas pelo padrão C ANSI (_DATE_, _FILE_, _LINE_, _TIME_ e _STDC_), além de oferecer mais sete macros (_18CXX, _nomedoprocessador, _SMALL_, _LARGE_, _TRADITIONAL18_, _EXTENDED18_).

4.17.1 _DATE_

Informa a data referente à última compilação do código.

Exemplo

//Suponha que o código tenha sido compilado no dia 18/04/2008.
`#include <stdio.h>` //Adiciona a biblioteca padrão de entrada e saída.

```

void main( ) //Função principal
{
    printf(_DATE_); //Imprime a mensagem: Apr 18 2008
}

```

4.17.2 _FILE_

Informa o nome do arquivo compilado.

Exemplo

//Suponha que o nome do arquivo compilado seja teste.c.
`#include <stdio.h>` //Adiciona a biblioteca padrão de entrada e saída.

```

void main( ) //Função principal
{
    printf(_FILE_); //Imprime a mensagem: teste.c
}

```

4.17.3 _LINE_

Informa a linha em que se encontra a macro _LINE_ que pode ser utilizada para localizar a linha correspondente a um determinado erro.

Exemplo

1. `#include<p18f4550.h>` //Arquivo de cabeçalho do PIC18F4550.
2. `#include <stdio.h>` //Adiciona a biblioteca padrão de entrada e saída.
- 3.
4. `void main()` //Função principal
5. {
6. `unsigned int tes;`

```

7.     tes=__LINE__;
8.
9.     fprintf( stdout, "Erro na linha - %u", tes); //Imprime a mensagem: Erro na linha - 7
10. }
```

4.17.4 __TIME__

A macro **__TIME__** informa o horário em que ocorreu a compilação do programa.

Exemplo

```

//Suponha que o arquivo tenha sido compilado às 20:40:10h.
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <stdio.h>//Adiciona a biblioteca padrão de entrada e saída.
```

```

void main( ) //Função principal
{ fprintf( stdout, "H Compilação - %S", __TIME__ ); } //Imprime a mensagem: H Compilação – 20:40.10
```

4.17.5 __STDC__

Verifica se o programa segue o padrão C ANSI. Se ele segue o padrão, a macro **__STDC__** retorna o valor 1. Qualquer outro valor diferente de 1 indica que o programa não segue o padrão C ANSI.

Exemplo

```

#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <stdio.h>//Adiciona a biblioteca padrão de entrada e saída.
```

```

void main( ) //Função principal
{ fpnntf( stdout, "\n0 - Nao Padrao C ANSI\n1 - Padrao C ANSI\n->%u", __STDC__ ); }
```

Mensagem impressa

0 - Nao Padrao C ANSI.
1 - Padrao C ANSI
→1

4.17.6 __18CXX__

Assume o valor 1 para indicar o compilador MPLAB® C18.

4.17.7 __Nomeprocessador

Assume o valor 1 se for compilado para um determinado processador. Por exemplo, **__18F4550** será definido como 1 caso seja compilado com a opção do *command-line* **-p18f4550**.

Sintaxe

__Nomeprocessador

Sendo:

- **Nomeprocessador**: nome do modelo do PIC®.

4.17.8 SMALL

Assume o valor 1, se for compilado com a opção do *command-line* *-ms*. A opção *-ms* define o modelo de memória do tipo pequeno (16bits).

Síntaxe

SMALL

4.17.9 LARGE

Assume o valor 1, se for compilado com a opção do *command-line* *-ml*. A opção *-ml* define o modelo de memória do tipo largo (24bits).

Síntaxe

LARGE

4.17.10 TRADITIONAL18

Assume o valor 1, se o compilador estiver usando o modo não estendido.

Síntaxe

TRADITIONAL18

4.17.11 EXTENDED18

Assume o valor 1, se o compilador estiver usando o modo estendido.

Síntaxe

EXTENDED18

4.18 Funções de Saída de Caracteres

As funções de saída de caracteres (Apêndice A) são utilizadas para enviar dados (formatados ou não) para um periférico, *buffer* ou qualquer tipo de dispositivo de I/O. A biblioteca padrão *stdio.h* do MPLAB® C18 define dois destinos de saída (*_H_USER* e *_H_USART*) e duas *streams* (*stdout* e *stderr*), que estão sempre abertas e prontas para serem usadas. A utilização das funções listadas neste tópico requer a introdução da diretiva `#include <stdio.h>` dentro do código de programa, conforme o modelo apresentado pela função *putc*.

- ***_H_USER*:** envio dos dados através da função de saída definida pelo usuário (*_user_putc*). Somente a estrutura da função está montada, cabendo ao programador criar o bloco de código.
- ***_H_USART*:** envio dos dados através da função de saída da biblioteca (*_uart_putc*). Ela também é usada para enviar dados para o periférico USART ou USART1, caso o dispositivo possua mais de um módulo USART.

O destino de saída associado às *streams* *stdout* e *stderr* é por definição o *_H_USART*, e pode ser modificado pelas seguintes instruções:

```
stdout = _H_USER;
stderr = _H_USER;
stdout = _H_USART;
stderr = _H_USART;
```

A partir deste momento veremos as funções de envio de caracteres presentes na biblioteca `stdio.h`. Podemos classificá-las como `put` e `printf`. As funções `put` enviam caracteres não formatados e suportam o comando `\n` (nova linha), enquanto as funções `printf` suportam o envio de caracteres formatados, resultando em um código relativamente maior que o `put`. Por este motivo as funções `put` são frequentemente usadas quando se deseja otimizar o código.

Caso tenha dificuldades em visualizar os caracteres impressos na janela **Output** do programa MPLAB® IDE, veja como configurar a UART do simulador MPLAB® SIM, no Capítulo 2.

4.18.1 Putc

Envia um caractere para uma determinada *stream* de saída. Se o envio for bem-sucedido, a função retorna o caractere enviado; caso contrário, retorna **EOF** (*End of File*).

Sintaxe

`res = putc (caractere, stream)`

Sendo:

- **res**: valor do tipo `int`.
- **caractere**: constante do tipo `char`.
- **stream**: é a *stream* de saída.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída.
```

```
void main (void)
{
    unsigned char caractere='P';
    stdout = _H_USART; //Configura o destino de saída.

    putc(caractere,stdout); //Envia o caractere 'P' para o destino de saída.
    putc('l',stdout); //Envia o caractere 'l' para o destino de saída.
    putc('\n',stdout); //Envia o comando de nova linha.
}
```

4.18.2 Puts

Envia uma *string* para uma `stdout` por sucessivas chamadas da função `putc` e anexa o comando de nova linha (`\n`) no final da *string*. Se o envio for bem-sucedido, a função retorna um valor positivo; caso contrário, retorna **EOF** (*End of File*).

Sintaxe

`res = puts (string)`

Sendo:

- **res**: valor do tipo `int`.
- **string**: mensagem a ser enviada.

Exemplo

```
puts("TESTE COMANDO PUTS"); //Envia a string para o stdout e adiciona um comando de nova linha ao final da string.
```

4.18.3 Fputs

Envia uma *string* para uma determinada *stream* de saída por sucessivas chamadas da função *putc* e anexa o comando de nova linha (*\n*) ao final da *string*. Se o envio for bem-sucedido, a função retorna um valor positivo; caso contrário, retorna **EOF** (*End of File*).

Sintaxe

```
res = fputs (string, stream)
```

Sendo:

- **res**: valor do tipo *int*.
- **string**: mensagem a ser enviada.
- **stream**: é a *stream* de saída.

Exemplo

```
stdout = _H_USART; //Configura o destino de saída.  
//Envia a string para o stdout e adiciona um comando de nova linha ao final da string.  
fputs("TESTE COMANDO FPUTS",stdout);
```

4.18.4 Printf

A função *printf* envia a *string* formatada para uma *stdout* através de sucessivas chamadas da função *putc*. Os caracteres que compõem a *string* são processados um de cada vez e enviados como aparecem na *string*, exceto para formatos específicos. Um formato específico é indicado dentro da *string* pelo prefixo *%* seguido dos componentes listados nas tabelas a seguir. Este prefixo informa ao compilador que a variável deve ser formatada antes de ser transmitida.

Se o envio for bem-sucedido, a função retorna o número de caracteres enviados; caso contrário, retorna **EOF** (*End of File*).

Sintaxe

```
res = printf (string, variáveis )
```

Sendo:

- **res**: valor do tipo *int*.
- **string**: mensagem a ser enviada.
- **variáveis**: variáveis do programa ou qualquer tipo de dado, separados por vírgula. (opcional)

A transmissão de variáveis do programa ou qualquer tipo de dado pode ser realizada adicionando o prefixo *"%"*, seguido do formato e tipo de dado, a uma *string*.

Sintaxe

```
%<suporte><tamanho><especificação_t><conversão>
```

Tabela 4.13: Parâmetro suporte.

Suporte (opcional)	Descrição
#	Modo alternativo de representar o dado. o - Adiciona um 0 antes do resultado. x - Adiciona o prefixo 0x. X - Adiciona o prefixo 0X. b - Adiciona o prefixo 0b. B - Adiciona o prefixo 0B.
-	Informa ao compilador que o resultado deve ser justificado para a esquerda. Se não for especificado, o compilador justifica o resultado para a direita.
+	Informa ao compilador que o resultado deve ser sinalizado com + ou -. Se não for especificado, o resultado só apresenta sinal, se for negativo. Válido somente para o formato sinalizado (d ou i).
0	Determina que o dado transmitido deve ser completado com '0' caso seja menor que o valor especificado pelo parâmetro tamanho; caso contrário, será preenchido com espaço '. Válido somente para o formato inteiro (d, i, o, u, b, B, x, X).

Tabela 4.14: Parâmetro tamanho.

Tamanho (opcional)	Descrição
tamanho	Define a quantidade mínima de caracteres que deve ser transmitida.
tamanho.cd	O parâmetro <i>tamanho</i> define a quantidade mínima de caracteres que deve ser transmitida e <i>cd</i> a quantidade mínima de dígitos. Válido somente para o formato inteiro (d, i, o, u, b, B, x, X).

Tabela 4.15: Parâmetro especificação_t.

Especificação_t (opcional)	Descrição
hh	Para especificadores de conversão do tipo inteiro, o argumento a ser convertido é um <i>signed char</i> ou <i>unsigned char</i> e o especificador determina um ponteiro para um argumento <i>signed char</i> .
h	Para especificadores de conversão do tipo inteiro, o argumento a ser convertido é um <i>short int</i> ou <i>unsigned short int</i> e o especificador determina um ponteiro para um argumento <i>short int</i> .
H	Para especificadores de conversão do tipo inteiro, o argumento a ser convertido é um <i>short long int</i> ou <i>unsigned short long int</i> e o especificador determina um ponteiro para um argumento <i>short long int</i> .
j	Para especificadores de conversão do tipo inteiro, o argumento a ser convertido é um <i>intmax_t</i> ou <i>uintmax_t</i> e o especificador determina um ponteiro para um argumento <i>intmax_t</i> . É equivalente ao especificador de tamanho l.
l	Para especificadores de conversão do tipo inteiro, o argumento a ser convertido é um <i>long int</i> ou <i>unsigned long int</i> e o especificador determina um ponteiro para um argumento <i>long int</i> .
t	Para especificadores de conversão do tipo inteiro, o argumento a ser convertido é um <i>ptrdiff_t</i> e o especificador determina um ponteiro para um argumento <i>ptrdiff_t</i> . É equivalente ao especificador de tamanho h.
T	Para especificadores de conversão do tipo inteiro, o argumento a ser convertido é um <i>ptrdifffrom_t</i> e o especificador determina um ponteiro para um argumento <i>ptrdifffrom_t</i> . É equivalente ao especificador de tamanho H.

Especificação_t (opcional)	Descrição
z	Para especificadores de conversão do tipo inteiro, o argumento a ser convertido é um <code>size_t</code> e o especificador determina um ponteiro para um argumento <code>size_t</code> . É equivalente ao especificador de tamanho <code>h</code> .
Z	Para especificadores de conversão do tipo inteiro, o argumento a ser convertido é um <code>sizerom_t</code> e o especificador determina um ponteiro para um argumento <code>sizerom_t</code> . É equivalente ao especificador de tamanho <code>H</code> .

Tabela 4.16: Parâmetro conversão.

Conversão	Descrição	Variável de entrada
c	Caractere.	<code>unsigned char</code>
d, i	Inteiro com sinal.	<code>signed int</code>
o	Inteiro em formato octal sem sinal.	<code>unsigned int</code>
b	Inteiro em formato binário sem sinal.	<code>unsigned int</code>
B	Inteiro em formato binário sem sinal.	<code>unsigned int</code>
s	Os elementos da matriz de caracteres são enviados até que seja encontrada uma terminação <code>\0</code> , ou até que a quantidade de elementos enviados seja igual ao <i>tamanho</i> especificado.	<code>char</code>
S	Funciona da mesma forma que o s, porém para <code>far rom char *</code> deve-se usar o especificador de tamanho <code>'H'</code> .	<code>char</code>
u	Inteiro sem sinal.	<code>unsigned int</code>
x	Inteiro em formato hexadecimal com letras minúsculas.	<code>unsigned int</code>
X	Inteiro em formato hexadecimal com letras maiúsculas	<code>unsigned int</code>

Exemplo

```

far rom char * mensagem = "Seja bem-vindo!!!";
unsigned char matriz[ ]={"TESTE"};
unsigned int num_1=29382;
signed int num_2=-29320;
unsigned short long num_3=0x88322;
char num_4=143;

fprintf (_H_USART, "\nTestando a UART....");
fprintf (stdout, "\nString na ROM -> %HS", mensagem);
fprintf (stdout, "\nString na RAM -> %s", matriz);
printf ("Valor do tipo unsigned int -> %07u", num_1);
printf ("Valor do tipo signed int -> %d", num_1);
printf ("\nValor do tipo signed int -> %+03d", num_2);

fprintf (stdout, "\nValor inteiro em formato hexadecimal -> %#hx", num_3);
fprintf (stdout, "\nValor inteiro em formato hexadecimal -> %#HX", num_3);
fprintf (stderr, "\nValor inteiro em formato binário -> %#hhb", num_4);
fprintf (stderr, "\nValor inteiro em formato binário -> %#hhB", num_4);

```

Mensagem impressa

Testando a UART....

String na ROM -> Seja bem-vindo!!!

String na RAM -> TESTE

Valor do tipo unsigned int -> 0029382
 Valor do tipo signed int -> +29382
 Valor do tipo signed int -> -029320
 Valor inteiro em formato hexadecimal -> 0x88322
 Valor inteiro em formato hexadecimal -> 0X88322
 Valor inteiro em formato binário -> 0b10001111
 Valor inteiro em formato binário -> 0B10001111

4.18.5 Fprintf

Essa função é similar à função `printf`, no entanto os dados são enviados para uma determinada *stream* de saída.

Sintaxe

`res = fprintf (stream, string, variáveis)`

Sendo:

- **res**: valor do tipo `int`.
- **stream**: é a *stream* de saída.
- **string**: mensagem a ser enviada.
- **variáveis**: variáveis do programa ou qualquer tipo de dado, separados por vírgula. (opcional)

4.18.6 Sprintf

Essa função é similar à função `printf`, no entanto os dados são enviados para uma determinada *buffer* de saída.

Sintaxe

`res = sprintf (buffer, string, variáveis)`

Sendo:

- **res**: valor do tipo `int`.
- **buffer**: buffer para recepção dos dados.
- **string**: mensagem a ser enviada.
- **variáveis**: variáveis do programa ou qualquer tipo de dado, separados por vírgula. (opcional)

Exemplo

```
unsigned char matriz_rec[15];
int temperatura=32;
sprintf(matriz_rec,"Temperatura=%03u", temperatura);//Envia a string. matriz_rec="Temperatura=032"
```

4.18.7 Vprintf

Essa função é similar à função `printf`, porém ela imprime uma lista de argumentos de variáveis no *stdout*.

Sintaxe

`res = vprintf (string, variáveis)`

Sendo:

- **res**: valor do tipo **int**.
- **string**: mensagem a ser enviada.
- **variáveis**: variáveis do programa ou qualquer tipo de dado, separados por vírgula. (opcional)

Exemplo

```
unsigned int matriz[5]={1,10,100,1000,10000};
va_list list_1;//Declara uma lista de argumentos.
va_start(list_1,matriz[5]);//Inicia a lista de argumentos a partir do ponto indicado.
va_end(list_1);//Finaliza a lista de argumentos.
va_start(list_1,matriz[5]);//Inicia a lista de argumentos a partir do ponto indicado.

//Envia a string formatada para o stdout.
vprintf("\nTESTE COMANDO VPRINTF\n4-%u\n3-%u\n2-%u\n1-%u\n0-%u",list_1);
```

Mensagem impressa

```
TESTE COMANDO VPRINTF
4-10000
3-1000
2-100
1-10
0-1
```

O exemplo anterior também pode ser implementado usando o **printf** e **va_arg**. Veja em seguida.

```
unsigned int matriz[5]={1,10,100,1000,10000};
va_list list_1;//Declara uma lista de argumentos.
va_start(list_1,matriz[5]);//Inicia a lista de argumentos a partir do ponto indicado.
va_end(list_1);//Finaliza a lista de argumentos.
va_start(list_1,matriz[5]);//Inicia a lista de argumentos a partir do ponto indicado

printf("\nTESTE COMANDO VPRINTF\n");//Envia a string para o stdout.
printf("4-%u\n",va_arg(list_1,int));//Envia o valor presente na posição 4 da matriz.
printf("3-%u\n",va_arg(list_1,int));//Envia o valor presente na posição 3 da matriz.
printf("2-%u\n",va_arg(list_1,int));//Envia o valor presente na posição 2 da matriz.
printf("1-%u\n",va_arg(list_1,int));//Envia o valor presente na posição 1 da matriz.
printf("0-%u\n",va_arg(list_1,int));//Envia o valor presente na posição 0 da matriz.
```



Os valores dos argumentos passados para a lista de argumentos sofrem conversão de tipo de dado.
Por exemplo, **float** – **double**, **unsigned/signed char/short** – **int** e **unsigned short** – **unsigned int**.

4.18.8 Vfprintf

Essa função é similar à função **printf**, porém ela imprime uma lista de argumentos de variáveis na *stream*.

Sintaxe

```
res = vfprintf (stream, string, variáveis )
```

Sendo:

- **res**: valor do tipo **int**.
- **stream**: é a *stream* de saída.

- **string:** mensagem a ser enviada.
- **variáveis:** variáveis do programa ou qualquer tipo de dado, separados por vírgula. (opcional)

Exemplo

//Suponha que list_1 é uma lista de argumentos.

```
vfprintf( stderr, "\nTESTE COMANDO VFPRINTF\n4-%u\n3-%u\n2-%u\n1-%u\n0-%u",list_1 );
```

4.18.9 Vsprintf

Essa função é similar à função `printf`, porém ela imprime uma lista de argumentos de variáveis no *buffer*.

Sintaxe

```
res = vsprintf ( stream, string, variáveis )
```

Sendo:

- **res:** valor do tipo `int`.
- **buffer:** buffer para recepção dos dados.
- **string:** mensagem a ser enviada.

Exemplo

//Suponha que list_1 é uma lista de argumentos.

```
vsprintf( matriz_res,"nTESTE COMANDO VSPRINTF\n4-%u\n3-%u\n2-%u\n1-%u\n0-%u",list_1 );
```

4.18.10 _Usart_Putc

Essa função envia um caractere pela USART. Se o dispositivo possuir mais do que uma USART, por definição será utilizada a USART1.

Sintaxe

```
_usart_putc ( caractere )
```

Sendo:

- **caractere:** constante do tipo `char`.

Exemplo

```
_usart_putc('O'); //Envia o caractere 'O' para a USART.
```

```
_usart_putc('K'); //Envia o caractere 'K' para a USART.
```

```
_usart_putc("\n"); //Envia o comando de "nova linha" para a USART.
```

4.18.11 _User_Putc

Essa função envia um caractere para uma aplicação definida pelo usuário.

Sintaxe

```
_user_putc ( caractere )
```

Sendo:

- **caractere:** constante do tipo `char`.

Exemplo

```
_user_putc('O'); //Envia o caractere 'O' para uma aplicação definida pelo usuário.
```

4.19 Funções Diversas

4.19.1 Funções de Manipulações de Bit/Byte

Esta seção apresenta as principais funções de manipulação de *bit/byte*.

4.19.1.1 RLNCF E RRNCF

A função **Rlncf** rotaciona a variável *var* para a esquerda, enquanto a função **Rrncf** rotaciona para a direita, Figura 4.3.

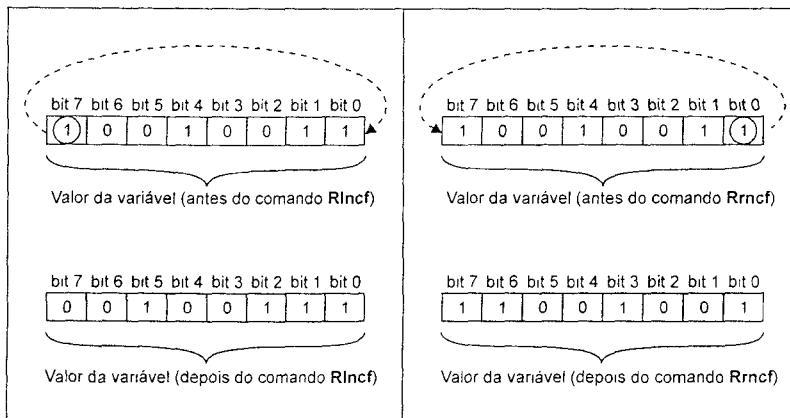


Figura 4.3: Operações realizadas pelos comandos **Rlncf** e **Rrncf**.

Sintaxe

Rlncf (*var, destino, Access*)

Rrncf (*var, destino, Access*)

Sendo:

- **var**: variável de 8*bits* não localizada na pilha.
- **destino** 0: o resultado é armazenado no **WREG**.
1: o resultado é armazenado na variável.
- **Access** 0: o *Access Bank* é selecionado.
1: o banco é selecionado por **BSR**.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
```

```
#pragma udata access minha_ram = 0x00
```

```
    near char var_access; //Aloca uma variável do tipo char, dentro da Access RAM. [0x0000]
```

```
#pragma udata
```

```
#pragma udata minha_ram_bs=0x200
```

```
    char var_banco; //Aloca uma variável do tipo char, dentro da No-Access RAM. [0x0200]
```

```
#pragma udata

void main( ) //Função principal
{
    var_access=0b01001111;
    var_banco=0b11000001;

    Rlncf(var_banco,1,1); //var_banco = 10000011
    Rlncf(var_banco,1,1); //var_banco = 00000111
    Rlncf(var_banco,1,1); //var_banco = 00001110
    Rrncf(var_banco,1,1); //var_banco = 00000111
    Rrncf(var_banco,1,1); //var_banco = 10000011
    Rrncf(var_banco,1,1); //var_banco = 11000001

    Rlncf(var_access,1,0); //var_access = 10011110
    Rlncf(var_access,1,0); //var_access = 00111101
    Rrncf(var_access,1,0); //var_access = 10011110
    Rrncf(var_access,1,0); //var_access = 01001111
}
```

4.19.1.2 RLCF E RRCF

As funções **Rlcf** e **Rrcf** são semelhantes à **Rlncf** e **Rrncf**. A principal diferença é que as funções **Rlcf** e **Rrcf** rotacionam a variável *var* com o *Carry bit* (C) localizado no registrador de *Status*.

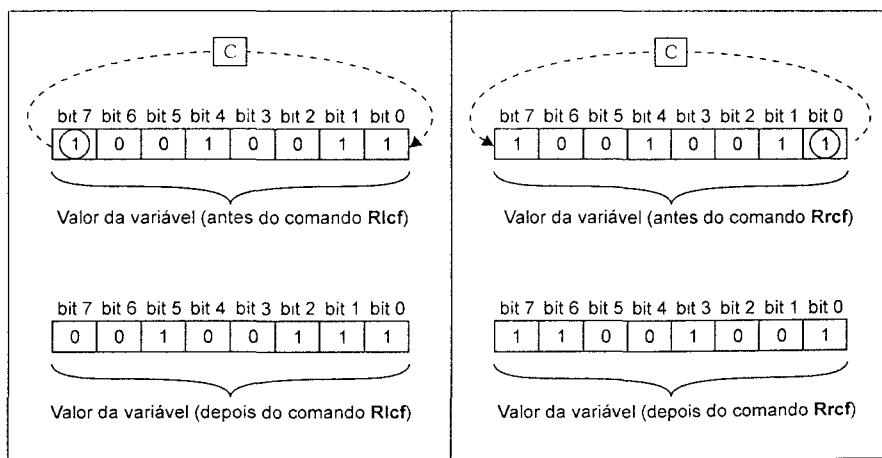


Figura 4.4: Operações realizadas pelos comandos **Rlcf** e **Rrcf**.

Sintaxe

Rlcf (*var, destino, Access*)
Rrcf (*var, destino, Access*)

Sendo:

- **var**: variável de 8bits não localizada na pilha.

- destino 0: o resultado é armazenado no **WREG**.
1: o resultado é armazenado na variável.
- **Access 0:** o Access Bank é selecionado.
1: o banco é selecionado por **BSR**.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.

#pragma udata access minha_ram = 0x00
    near char var_access; //Aloca uma variável dq tipo char dentro da Access RAM. [0x0000]
#pragma udata

#pragma udata minha_ram_bs=0x200
    char var_banco; //Aloca uma variável do tipo char dentro da No-Access RAM [0x0200]
#pragma udata

void main( ) //Função principal
{
    unsigned char x,y,res;

    x=200; y=200;

    var_access=0b01000001;
    var_banco=0b11000001;

    res = x+y; //res = 144;
    //Carry bit = 1, pois na operação de adição vai 1 para fora
    //400 = 1 1001 0000
    //res = 1001 0000
    Rlcf(var_banco,1,1); //var_banco = 1000 0011. Após esta operação, o Carry bit = 1, pois sai 1.
    Rlcf(var_banco,1,1); //var_banco = 0000 0111. Após esta operação, o Carry bit = 1, pois sai 1.
    Rlcf(var_banco,1,1); //var_banco = 0000 1111. Após esta operação, o Carry bit = 0, pois sai 0.
    Rlcf(var_banco,1,1); //var_banco = 0001 1110. Após esta operação, o Carry bit = 0, pois sai 0.
    //Carry bit = 0
    Rrcf(var_access,1,0); //var_access = 0010 0000. Após esta operação, o Carry bit = 1, pois sai 1.
    Rrcf(var_access,1,0); //var_access = 1001 0000. Após esta operação, o Carry bit = 0, pois sai 0.
    Rrcf(var_access,1,0); //var_access = 0100 1000. Após esta operação, o Carry bit = 0, pois sai 0.
}
```

4.19.1.3 SWAPF

O comando **Swapf** inverte o nible mais significativo pelo menos significativo de uma variável inteira de 8bits.

Sintaxe

Swapf (*var, destino, Access*)

Sendo:

- **var:** variável de 8bits não localizada na pilha.
- **destino 0:** o resultado é armazenado no **WREG**.
1: o resultado é armazenado na variável.
- **Access 0:** o Access Bank é selecionado.
1: o banco é selecionado por **BSR**.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.

#pragma udata access minha_ram = 0x00
    near char var_access; //Aloca uma variável do tipo char, dentro da Access RAM. [0x0000]
#pragma udata
#pragma udata minha_ram_bs=0x200
    char var_banco; //Aloca uma variável do tipo char, dentro da No-Access RAM. [0x0200]
#pragma udata

void main( ) //Função principal
{
    var_access=0b01001111;
    var_banco=0b11000001;

    Swapf(var_banco,1,1); //var_banco = 0001 1100
    Swapf(var_access,1,0); //var_access = 1111 0100
}
```

4.19.2 Funções de Classificação de Caracteres

As funções utilizadas para manipulação de caracteres se encontram no arquivo **ctype.h**. A utilização das funções listadas neste tópico requer a introdução da diretiva `#include <ctype.h>` dentro do código de programa, conforme o modelo apresentado pela função **isalnum**

4.19.2.1 Isalnum

A função **isalnum** verifica se um determinado *caractere* pertence ao conjunto {'0':'9'}, {'A':'Z'} ou {'a':'z'}. Caso ele pertença, a função retorna o valor 1; caso contrário, 0.

Sintaxe

resultado = **isalnum** (*caractere*)

Sendo:

- **caractere**: variável do tipo **char**.
- **resultado**: valor booleano.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include<ctype.h> //Adiciona a biblioteca que contém as funções de manipulação de caracteres.
```

```
void main (void)
{
    char caractere = 'c'; //Declara uma variável do tipo char com atribuição de valor.
    short res; //Declara uma variável do tipo short.
    res = isalnum ( caractere ); //res = 1, pois o caractere 'c' pertence ao conjunto predeterminado.
}
```

4.19.2.2 Isalpha

A função **isalpha** verifica se um determinado *caractere* pertence ao conjunto {'A':'Z'} ou {'a':'z'}. Caso ele pertença, a função retorna o valor 1; caso contrário, 0.

Sintaxe

```
resultado = isalpha ( caractere )
```

Sendo:

- **caractere**: variável do tipo **char**.
- **resultado**: valor booleano.

Exemplo

```
char caractere = '1'; //Declara uma variável do tipo char com atribuição de valor.  
short res; //Declara uma variável do tipo short  
res = isalpha ( caractere ); //res = 0, pois o caractere '1' não pertence ao conjunto predeterminado.
```

4.19.2.3 Isdigit

A função **isdigit** verifica se um determinado *caractere* pertence ao conjunto {'0':'9'}. Caso ele pertença, a função retorna o valor 1; caso contrário, 0.

Sintaxe

```
resultado = isdigit ( caractere )
```

Sendo:

- **caractere**: variável do tipo **char**.
- **resultado**: valor booleano.

Exemplo

```
char caractere = '4'; //Declara uma variável do tipo char com atribuição de valor.  
char res; //Declara uma variável do tipo char.  
res = isdigit('c'); //res = 0.  
res = isdigit('1'); //res = 1.  
res = isdigit(caractere), //res = 1.
```

4.19.2.4 Islower

A função **islower** verifica se um determinado *caractere* pertence ao conjunto {'a':'z'}. Caso ele pertença, a função retorna o valor 1; caso contrário, 0.

Sintaxe

```
resultado = islower ( caractere )
```

Sendo:

- **caractere**: variável do tipo **char**.
- **resultado**: valor booleano.

Exemplo

```
char res; //Declara uma variável do tipo char.  
res = islower('c'); // res = 1.  
res = islower('C'); //res = 0.
```

4.19.2.5 isspace

A função **isspace** verifica se um determinado *caractere* é um espaço (' '). Caso ele seja, a função retorna o valor 1; caso contrário, 0.

Sintaxe

resultado = **isspace** (*caractere*)

Sendo:

- **caractere**: variável do tipo **char**.
- **resultado**: valor booleano.

Exemplo

```
char caractere = ' '; //Declara uma variável do tipo char com atribuição de valor.
char res; //Declara uma variável do tipo char.
res = isspace(caractere); //res = 1.
```

4.19.2.6 isupper

A função **isupper** verifica se um determinado *caractere* pertence ao conjunto {A:'Z'}. Caso ele seja, a função retorna o valor 1; caso contrário, 0.

Sintaxe

resultado = **isupper** (*caractere*)

Sendo:

- **caractere**: variável do tipo **char**.
- **resultado**: valor booleano.

Exemplo

```
char caractere_1 = 't'; //Declara uma variável do tipo char com atribuição de valor.
char caractere_2 = 'T'; //Declara uma variável do tipo char com atribuição de valor.
char res; //Declara uma variável do tipo char.
```

```
res = isupper(caractere_1); //res = 0.
res = isupper(caractere_2); //res = 1.
```

4.19.2.7 isxdigit

A função **isxdigit** verifica se um determinado *caractere* pertence ao conjunto {0:'9'}, {a:'f'} ou {A:'F'}. Caso ele seja, a função retorna o valor 1; caso contrário, 0.

Sintaxe

resultado = **isxdigit** (*caractere*)

Sendo:

- **caractere**: variável do tipo **char**.
- **resultado**: valor booleano.

Exemplo

```
char caractere_1 = 'f'; //Declara uma variável do tipo char com atribuição de valor.
char caractere_2 = 'g'; //Declara uma variável do tipo char com atribuição de valor.
char res; //Declara uma variável do tipo char.
```

```
res = isxdigit(caractere_1); //res = 1.
res = isxdigit(caractere_2); //res = 0
```

4.19.2.8 Towlower

A função **towlower** transforma o *caractere* em minúsculo.

Sintaxe

```
resultado = tolower ( caractere )
```

Sendo:

- **caractere**: variável do tipo **char**.
- **resultado**: caractere de 8bits.

Exemplo

```
unsigned char caractere_1='F', caractere_2='G'; //Declara duas variáveis do tipo char com atribuição de valor.
unsigned char res; //Declara uma variável do tipo boolean.
```

```
res = tolower(caractere_1); //res = 'f'.
res = tolower(caractere_2); //res = 'g'.
```

4.19.2.9 Toupper

A função **toupper** transforma o *caractere* em maiúsculo.

Sintaxe

```
resultado = toupper ( caractere )
```

Sendo:

- **caractere**: variável do tipo **char**.
- **resultado**: caractere de 8bits.

Exemplo

```
unsigned char caractere_1 = 'k'; //Declara uma variável do tipo char com atribuição de valor.
unsigned char caractere_2 = 'h'; //Declara uma variável do tipo char com atribuição de valor.
unsigned char res; //Declara uma variável do tipo boolean.
```

```
res = toupper(caractere_1); //res = 'K'.
res = toupper(caractere_2); //res = 'H'.
```

4.19.2.10 Iscntrl

Verifica se um caractere pertence aos caracteres de controle. Um caractere de controle não é imprimível, logo deve estar no intervalo [0x00 : 0x1F] ou igual a 0x7F. A função **iscntrl** retorna um valor diferente de zero se o caractere for de controle; caso contrário, retorna 0.

Sintaxe

```
valor = iscntrl ( caractere )
```

Sendo:

- **valor:** valor do tipo `unsigned char`.
- **caractere:** valor do tipo `unsigned char`.

Exemplo

```
unsigned char valor;
```

```
valor= iscntrl (0x00); //valor = 1
valor= iscntrl ('@'); //valor = 0
valor= iscntrl ('\n'); //valor = 1
valor= iscntrl (0x8F); //valor = 0
```

4.19.2.11 isgraph

Determina se um caractere é gráfico. Um caractere gráfico é imprimível, com exceção do espaço ". A função `isgraph` retorna um valor diferente de zero se o caractere pertencer ao intervalo; caso contrário, retorna 0.

Sintaxe

```
valor = isgraph ( caractere )
```

Sendo:

- **valor:** valor do tipo `unsigned char`.
- **caractere:** valor do tipo `unsigned char`.

Exemplo

```
unsigned char valor;
```

```
valor= isgraph ('3'); //valor = 1
valor= isgraph ('\n'); //valor = 0
valor= isgraph ('!'); //valor = 1
```

4.19.2.12 isprint

Verifica se um determinado caractere é imprimível ou não. Um caractere é considerado imprimível se o valor correspondente a ele estiver no intervalo [0x20 : 0x7E]. A função `isprint` retorna um valor diferente de zero se o caractere pertencer ao intervalo; caso contrário, retorna 0.

Sintaxe

```
valor = isprint ( caractere )
```

Sendo:

- **valor:** valor do tipo `unsigned char`.
- **caractere:** valor do tipo `unsigned char`.

Exemplo

```
unsigned char valor;
```

```
valor=isprint ('T'); //valor = 1
valor=isprint ('*'); //valor = 1
valor=isprint ('\u03bc'); //valor = 0
```

4.19.2.13 `ispunct`

Verifica se um determinado caractere se enquadra como caractere de pontuação. Um caractere de pontuação deve ser imprimível e não deve ser um espaço ' ' ou qualquer caractere alfanumérico. A função `ispunct` retorna um valor diferente de zero se o caractere for de pontuação; caso contrário, retorna 0.

Sintaxe

```
valor = ispunct ( caractere )
```

Sendo:

- **valor**: valor do tipo `unsigned char`.
- **caractere**: valor do tipo `unsigned char`.

Exemplo

```
unsigned char valor;
```

```
valor=ispunct ('T'); //valor = 0
```

```
valor=ispunct ('a'); //valor = 0
```

```
valor=ispunct (' '); //valor = 0
```

4.19.3 Funções de Conversão de Dados

As funções utilizadas para manipulação de dados se encontram no arquivo `stdlib.h`. A utilização das funções listadas neste tópico requer a introdução da diretiva `#include < stdlib.h >` dentro do código de programa, conforme o modelo apresentado pela função `atob`.

4.19.3.1 `Atob`, `Atof`, `Atoi` e `Atol`

As funções `atob`, `atoi` e `atof` convertem uma *string* em um dado do tipo `signed char` (8bits), `signed int` (16bits) e `signed long` (32bits) respectivamente, enquanto a função `atof` converte a *string* em um valor do tipo `float`.

Sintaxe

```
valor_8bits = atob ( texto )
```

```
valor_16bits = atof ( texto )
```

```
valor_32bits = atoi ( texto )
```

```
valor_float = atol ( texto )
```

Sendo:

- **valor_8bits**: dado do tipo `signed char`.
- **valor_16bits**: dado do tipo `signed int`.
- **valor_32bits**: dado do tipo `signed long`.
- **valor_float**: dado do tipo `float`.
- **texto**: *string* contendo um valor inteiro ou `float`.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
```

```
#include<stdlib.h> //Adiciona a biblioteca de funções miscelâneas.
```

```
void main (void)
{
```

```

unsigned char mat_char[] = {"139"}; //Declara uma string.
unsigned char mat_long[] = {"483792"}; //Declara uma string.
unsigned char mat_float[] = {"142.76"}; //Declara uma string.

unsigned char valor_char; //Declara uma variável do tipo int (8bits).
long valor_long; //Declara uma variável do tipo long (32bits).
float valor_float; //Declara uma variável do tipo float (32bits).

valor_char = atob (mat_char); //valor_char = 139.
valor_long = atol (mat_long); //valor_long = 483792.
valor_long = atol (mat_float); //valor_long = 142.
valor_float = atof (mat_float); //valor_float = 142.76.
}

```

4.19.3.2 Btoa, Itoa, Ltoa e Ultoa

As funções **btoa**, **itoa** e **ltoa** convertem um dado do tipo **signed char** (8bits), **signed int** (16bits) e **signed long** (32bits), respectivamente, em uma **string**, enquanto a função **ultoa** converte um dado do tipo **unsigned long** em uma **string**.

Sintaxe

```

btoa ( valor_8bits, texto )
itoa ( valor_16bits, texto )
ltoa ( valor_s32bits, texto )
ultoa ( valor_u32bits, texto )

```

Sendo:

- **valor_8bits**: dado do tipo **signed char**.
- **valor_16bits**: dado do tipo **signed int**.
- **valor_s32bits**: dado do tipo **signed long**.
- **valor_u32bits**: dado do tipo **unsigned long**.
- **texto**: **string** que vai receber o valor convertido.

Exemplo

```

unsigned char mat_char[4]; //Declara uma string com 4 posições.
unsigned char mat_int[6]; //Declara uma string com 6 posições.
unsigned char mat_slong[11]; //Declara uma string com 11 posições.
unsigned char mat_ulong[11]; //Declara uma string com 11 posições.
signed char valor_char; //Declara uma variável do tipo signed char (8bits).
signed int valor_int; //Declara uma variável do tipo signed int(16bits)
signed long valor_slong; //Declara uma variável do tipo signed long(32bits).
unsigned long valor_ulong; //Declara uma variável do tipo unsigned long (32bits).

```

```

valor_char=-110;
valor_int=3928;
valor_slong=-922218;
valor_ulong=6482493;

```

```

btoa (valor_char, mat_char); //mat_char = "-110".
itoa (valor_int, mat_int); //mat_int = "3928".
ltoa (valor_slong, mat_slong); //mat_slong = "-922218".
ultoa (valor_ulong, mat_ulong); //mat_ulong = "6482493".

```

4.19.4 Funções de Manipulação de Memória e String

As funções de manipulação de memória e *string* estão implementadas no arquivo **string.h**. A utilização das funções listadas neste tópico requer a introdução da diretiva `#include < string.h >` dentro do código de programa, conforme o modelo apresentado pela função **memchr**.



As funções `*2pgm` não escrevem na memória flash, elas escrevem no barramento da memória de programa, disponível em alguns modelos do PIC18.

4.19.4.1 Memchr e Memchrpgm

A função **memchr** procura a primeira ocorrência de *c* (**unsigned char**) nos *n* primeiros bytes da região da memória de dados apontada por *s*, e retorna um ponteiro para o caractere; caso contrário, retorna um ponteiro nulo.

A função **memchrpgm** realiza as mesmas tarefas que a função anterior, porém ela faz a pesquisa na memória de programa.

Sintaxe

localização = **memchr** (*s*, *c*, *n*)
localização = **memchrpgm** (*s*, *c*, *n*)

Sendo:

- *s*: ponteiro para uma região da memória.
- *c*: valor do dado **unsigned char**.
- *n*: número de *bytes* para pesquisa.
- *localização*: ponteiro para o dado *c* dentro da memória.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.  

#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata overlay minha_rom=0x300  

    rom char mem_2[ ]={"Memoria de programa"};//Armazena a string na memória de programa. [0x0300 - 0x0313]  

#pragma romdata  
  

#pragma idata access minha_ram=0x00  

    char mem_1[ ]={"Memoria de dados"};//Armazena a string na memória de dados. [0x0000 - 0x0010]  

#pragma idata
```

```
void main( ) //Função principal
```

```
{
```

```
    char *ponteiro_ram;  

    far rom char *ponteiro_rom;
```

```
/*Procura a primeira ocorrência do caractere 'd' dentro de toda string mem_1, localizada na memória de dados.*/  

ponteiro_ram = memchr(mem_1,'d',sizeof(mem_1)); //ponteiro_ram = 0x0008
```

```
/*Procura a primeira ocorrência do caractere 'o' dentro das oito primeiras posições da string mem_2, localizada na memória de programa.*/
```

```
ponteiro_rom = memchrpgm(mem_2,'o',8); //ponteiro_rom = 0x0303
```

```
}
```

4.19.4.2 Memcmp, Memcmppgm, Memcmppgm2ram e Memcmpram2pgm

A função **memcmp** compara os *n* primeiros *bytes* do bloco de memória de dados apontado por *buf_1*, com os *n* primeiros *bytes* do bloco de memória de dados apontado por *buf_2* e retorna um valor que se enquadra em uma das condições listadas em seguida.

- < 0: o primeiro *byte* que não combina em ambos os blocos é menor no *buf_1* do que no *buf_2*.
- = 0: conteúdo do *buf_1* é igual ao do *buf_2*.
- > 0: o primeiro *byte* que não combina em ambos os blocos é maior no *buf_1* do que no *buf_2*.

A função **memcmppgm** faz a comparação de dois blocos localizados na memória de programa.

A função **memcmppgm2ram** faz a comparação de um bloco localizado na memória de dados (*buf_1*) e outro bloco na memória de programa (*buf_2*).

A função **memcmpram2pgm** faz a comparação de um bloco localizado na memória de dados (*buf_2*) e outro bloco na memória de programa (*buf_1*).

Sintaxe

```
relação_tamanho = memcmp ( buf_1, buf_2, n )
relação_tamanho = memcmppgm ( buf_1, buf_2, n )
relação_tamanho = memcmppgm2ram ( buf_1, buf_2, n )
relação_tamanho = memcmpram2pgm ( buf_1, buf_2, n )
```

Sendo:

- **buf_1**: ponteiro para bloco de memória.
- **buf_2**: ponteiro para bloco de memória.
- **n**: número de *bytes* para pesquisa.
- **relação_tamanho**: valor do tipo **signed char**.

Exemplo

```
const rom char mem_2[ ]={"Memoria de programa 1"};//Armazena a string na memória de programa.
rom char mem_3[ ]={"Memoria de programa 2"};//Armazena a string na memória de programa.
```

```
void main( ) //Função principal
{
    char mem_1[ ]={"Memoria de dados 1"};//Armazena a string na memória de dados.
    char mem_4[ ]={"Memoria de dados 2"};//Armazena a string na memória de dados.
    signed char valor;

    valor = memcmp ( mem_1, mem_4, 10);           //Valor = 0 -> 'e'-'e'=0
    valor = memcmp ( mem_1, mem_4, sizeof(mem_1)); //Valor = -1 -> '1'-'2'=-1
    valor = memcmppgm ( mem_2, mem_3, sizeof(mem_3) ); //Valor = -1 -> '1'-'2'=-1
    valor = memcmppgm2ram ( mem_1, mem_3, 11 ); //Valor = 0 -> '.'-'.'=0
    valor = memcmpram2pgm ( mem_2, mem_1, 7 ); //Valor = 0 -> 'a'-'a'=0
    valor = memcmpram2pgm ( mem_2, mem_1, sizeof(mem_2) ); //Valor = 12 -> 'p'-'d'=12
}
```

4.19.4.3 Memcpy, Memcpypgm, Memcpypgm2ram e Memcpypyram2pgm

A função **memcpy** copia os *n* primeiros *bytes* do bloco de memória de dados apontado por *fonte* e insere no bloco de memória de dados apontado por *destino*. Esta função retorna o ponteiro do *destino*. Se *fonte* e *destino* forem sobrepostos, o comportamento dessa função é indefinido.

- **Função memcpypgm:** ambos os ponteiros apontam para o bloco de memória de programa.
- **Função memcpypgm2ram:** o ponteiro *destino* aponta para o bloco de memória de dados e o ponteiro *fonte* para o bloco de memória de programa.
- **Função memcpyram2pgm:** o ponteiro *fonte* aponta para o bloco de memória de dados e o ponteiro *destino* para o bloco de memória de programa.

Sintaxe

```
ponteiro_d = memcpy ( destino, fonte, n )
ponteiro_d = memcpypgm ( destino, fonte, n )
ponteiro_d = memcpypgm2ram ( destino, fonte, n )
ponteiro_d = memcpyram2pgm ( destino, fonte, n )
```

Sendo:

- **destino:** ponteiro para uma região da memória.
- **fonte:** ponteiro para uma região da memória.
- **n:** número de bytes a serem copiados.
- **ponteiro_d:** ponteiro para o *destino*.

Exemp'o

```
#include<pic18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata overlay minha_rom=0x0400
    rom char mem_2[21]={"Compilador "};//Armazena a string na memória de programa.[0x0400 - 0x0414]
    rom char mem_3[ ]={"MPLAB C18."};//Armazena a string na memória de programa. [0x0415 - 0x041F]
#pragma romdata

#pragma idata access minha_ram=0x0000
    char mem_1[21]={"Compilador "};//Armazena a string na memória de dados [0x0000 - 0x0014]
    const char mem_4[ ]={"MPLAB C18."};//Armazena a string na memória de dados. [0x0015 - 0x001F]
#pragma idata

void main( ) //Função principal
{
    char *ponteiro_ram;
    ponteiro_ram = memcpy ( &mem_1[11], mem_4, 5); //mem_1 = "Compilador MPLAB" e ponteiro_ram = 0x000B

    //mem_1 = "Compilador MPLAB C18." e ponteiro_ram = 0x000B
    ponteiro_ram = memcpy ( mem_1+11, mem_4, sizeof(mem_4));

    //mem_1 = "MPLilador MPLAB C18." e ponteiro_ram = 0x0000
    ponteiro_ram = memcpypgm2ram ( mem_1, mem_3, 3 );
}
```

4.19.4.4 Memmove, Memmovepgm, Memmovepgm2ram e Memmoveram2pgm

A função **memmove** é similar à função **memcpy**, porém ela desempenha a tarefa corretamente mesmo se ocorrer sobreposição da região, como se existisse um *buffer* intermediário.

- **Função memmovepgm:** ambos os ponteiros apontam para o bloco de memória de programa.

- **Função memmovepgm2ram:** o ponteiro *destino* aponta para o bloco de memória de dados e o ponteiro *fonte* para o bloco da memória de programa.
- **Função memmoveram2pgm:** o ponteiro *fonte* aponta para o bloco de memória de dados e o ponteiro *destino* para o bloco de memória de programa.

Sintaxe

```
ponteiro_d = memmove ( destino, fonte, n )
ponteiro_d = memmovepgm ( destino, fonte, n )
ponteiro_d = memmovepgm2ram ( destino, fonte, n )
ponteiro_d = memmoveram2pgm ( destino, fonte, n )
```

Sendo:

- **destino:** ponteiro para uma região da memória.
- **fonte:** ponteiro para uma região da memória.
- **n:** número de *bytes* a serem deslocados.
- **ponteiro_d:** ponteiro para o *destino*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.

#pragma romdata overlay minha_rom=0x0400
    rom char mem_3[ ]={"012345"};//Armazena a string na memória de programa. [0x0400 - 0x0406]
    rom char mem_4[ ]={"6789"};//Armazena a string na memória de programa. [0x0407 - 0x040B]
#pragma romdata

#pragma idata access minha_ram=0x0000
    char mem_1[21]=(“ABCD”);//Armazena a string na memória de dados. [0x0000 - 0x0014]
    const char mem_2[ ]={"EFGH"};//Armazena a string na memória de dados. [0x0015 - 0x0019]
#pragma idata

void main( ) //Função principal
{
    char novo_caractere='l';
    char *ponteiro_ram;

    //mem_1 = "ABCDEFGH" e ponteiro_ram = 0x0004
    ponteiro_ram = memmove ( &mem_1[4], mem_2, sizeof(mem_2));
    //mem_1 = "ABCDEFGHI" e ponteiro_ram = 0x0008
    ponteiro_ram = memmove ( &mem_1[8], &novo_caractere, 1);
    //mem_1 = "ABCDEFGHI012345" e ponteiro_ram = 0x0009
    ponteiro_ram = memmovepgm2ram ( &mem_1[9], mem_3, sizeof(mem_3));
}
```

4.19.4.5 Memset e Memsetpgm

A função **memset** copia o caractere *c* (**unsigned char**) nos *n* primeiros *bytes* do bloco de memória de dado apontado por *destino*, e retorna um ponteiro para o *destino*.

- **Função memsetpgm:** ambos os ponteiros apontam para o bloco de memória de programa.

Sintaxe
`ponteiro_d = memset (destino, c, n)`
`ponteiro_d = memsetpgm (destino, c, n)`

Sendo:

- **destino**: ponteiro para uma região da memória.
- **c**: caractere que será copiado.
- **n**: quantidade de bytes do *destino* em que o caractere *c* será copiado.
- **ponteiro_d**: ponteiro para o *destino*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma idata access minha_ram=0x00
    char mem_1[5]={"TE TE"};//Armazena a string na memória de dados. {0x0000 - 0x0014}
#pragma idata

void main( ) //Função principal
{
    char *ponteiro_ram;
    ponteiro_ram = memset ( &mem_1[2], 'S', 1); //mem_1 = "TESTE" e ponteiro_ram = 0x0002
}
```

4.19.4.6 Strcat, Strcatpgm, Strcatpgm2ram e Strcatram2pgm

A função **strcat** faz uma cópia da *string* apontada por *fonte*, anexa-a no final da *string* apontada por *destino* e adiciona um caractere nulo à *string* resultante. Essa função retorna o ponteiro do *destino*.

- **Função strcatpgm**: ambos os ponteiros apontam para a *string* localizada na memória de programa.
- **Função strcatpgm2ram**: o ponteiro *destino* aponta para a *string* localizada na memória de dados e o ponteiro *fonte* para o bloco da memória de programa.
- **Função strcatram2pgm**: o ponteiro *fonte* aponta para a *string* localizada na memória de dados e o ponteiro *destino* para o bloco da memória de programa.

Sintaxe

```
ponteiro_d = strcat ( destino, fonte )
ponteiro_d = strcatpgm ( destino, fonte )
ponteiro_d = strcatpgm2ram ( destino, fonte )
ponteiro_d = strcatram2pgm ( destino, fonte )
```

Sendo:

- **destino**: ponteiro para *string*.
- **fonte**: ponteiro para *string*.
- **ponteiro_d**: ponteiro para o *destino*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata overlay minha_rom=0x450
```

```

rom char mem_3[10]={"0123456789"};//Armazena a string na memória de programa.[0x0450 - 0x0459]
#pragma romdata

#pragma idata access minha_ram=0x00
    char mem_1[21]={"NUMEROS "};//Armazena a string na memória de dados. [0x0000 - 0x0014]
    const char mem_2[ ]={"->"}..//Armazena a string na memória de dados. [0x0015 - 0x0018]
#pragma idata

void main( ) //Função principal
{
    char *ponteiro_ram;

    //mem_1 = "NUMEROS -> " e ponteiro_ram = 0x0000
    ponteiro_ram = strcat ( mem_1, mem_2);
    //mem_1 = "NUMEROS -> 0123456789" e ponteiro_ram = 0x0000
    ponteiro_ram = strcatpgm2ram ( mem_1, mem_3);
}

```

4.19.4.7 Strchr e Strchrpgm

A função **strchr** procura a primeira ocorrência de *c* (**unsigned char**) dentro da *string* localizada na memória de dados e apontada por *s*, e retorna um ponteiro para o caractere dentro da *string*. Se o caractere não for encontrado, a função retorna um ponteiro nulo.

A função **strchrpgm** é similar à função anterior, no entanto ela faz a pesquisa na *string* localizada na memória de programa.

Sintaxe

localização = **strchr** (*s*, *c*)
localização = **strchrpgm** (*s*, *c*)

Sendo:

- **s**: ponteiro para *string*.
- **c**: valor do dado **unsigned char**.
- **localização**: ponteiro para o dado *c* dentro da *string* apontada por *s*.

Exemplo

#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.

```

#pragma romdata overlay minha_rom=0x450
    rom char mem_3[ ]={"MEMORIA DE PROGRAMA"}; //Local armazenado [0x0450 - 0x0463]
#pragma romdata

#pragma idata access minha_ram=0x00
    char mem_1[ ]={"MEMORIA DE DADOS"};//Armazena a string na memória de dados. [0x0000 - 0x0010]
    unsigned char mem_2[ ]={23,64,34,87,230,200};//Armazena a string na memória de dados. [0x0011 - 0x0016]
#pragma idata

void main( ) //Função principal
{
    char *ponteiro_ram;
    char *ponteiro_rom;

```

```

/*Procura a primeira ocorrência do caractere 'D' dentro de toda string mem_1, localizada na memória de dados.*/
ponteiro_ram = strchr(mem_1,'D');//ponteiro_ram = 0x0008
/*Procura a primeira ocorrência do valor 87 dentro de toda string mem_2, localizada na memória de dados.*/
ponteiro_ram = strchr(mem_2,87);//ponteiro_ram = 0x0014
/*Procura a primeira ocorrência do caractere 'O' dentro da string mem_3, localizada na memória de programa.*/
ponteiro_rom = strchrgm(mem_3,'O');//ponteiro_rom = 0x0453
}

```

4.19.4.8 Strcmp, Strcmppgm, Strcmppgm2ram e Strcmpram2pgm

A função `strcmp` compara os dados da *string* localizada na memória de dados e apontada por *s_1*, com os dados da *string* localizada na memória de dados e apontada por *s_2*. Essa função retorna um valor que se enquadra em uma das condições listadas em seguida.

- < 0: o primeiro *byte* que não combina em ambas as *strings* é menor no *s_1* do que no *s_2*.
- = 0: conteúdo do *s_1* é igual ao do *s_2*.
- > 0: o primeiro *byte* que não combina em ambas as *strings* é maior no *s_1* do que no *s_2*.

A função `strcmppgm` faz a comparação de duas *strings* localizadas na memória de programa.

A função `strcmppgm2ram` faz a comparação de uma *string* localizada na memória de dados (*s_1*) e a outra na memória de programa (*s_2*).

A função `strcmpram2pgm` faz a comparação de uma *string* localizada na memória de dados (*s_2*) e a outra na memória de programa (*s_1*).

Sintaxe

```

relação_tamanho = strcmp ( s_1, s_2 )
relação_tamanho = strcmppgm ( s_1, s_2 )
relação_tamanho = strcmppgm2ram ( s_1, s_2 )
relação_tamanho = strcmpram2pgm ( s_1, s_2 )

```

Sendo:

- ***s_1***: ponteiro para *string*.
- ***s_2***: ponteiro para *string*.
- **relação_tamanho**: valor do tipo **signed char**.

Exemplo

```

const rom char mem_4[ ]={"Compilador C18"};//Armazena a string na memória de programa.
const rom char mem_5[ ]={"Compilador MPLAB C18"};//Armazena a string na memória de programa.
const rom char mem_6[ ]={"Compilador MPLAB C18"};//Armazena a string na memória de programa.

```

```

void main( ) //Função principal
{
    char mem_1[ ]={"Compilador C18"};//Armazena a string na memória de dados.
    char mem_2[ ]={"Compilador MPLAB C18"};//Armazena a string na memória de dados.
    char mem_3[ ]={"Compilador MPLAB C18"};//Armazena a string na memória de dados.

    char mem_7[ ]={12,234,32,34,21};//Armazena a string na memória de dados.
    char mem_8[ ]={12,234,32,53,21};//Armazena a string na memória de dados.

    signed char valor;

    valor = strcmp ( mem_1, mem_2); //Valor = -10 -> 'C'-'M' = -10
    valor = strcmp ( mem_2, mem_1); //Valor = +10 -> 'M'-'C' = +10

```

```

valor = strcmp ( mem_1, mem_3); //Valor = -10 -> 'C'-'M' = -10
valor = strcmp ( mem_2, mem_3); //Valor = 0 -> 'M'-'M' = 0
valor = strcmp ( mem_7, mem_8); //Valor = -19 -> 34-53 = -19
valor = strcmp ( mem_8, mem_7); //Valor = +19 -> 53-34 = +19
valor = strcmppgm ( mem_5, mem_4 );//Valor = +10 -> 'M'-'C' = +10
valor = strcmppgm ( mem_4, mem_5 );//Valor = -10 -> 'C'-'M' = -10
valor = strcmppgm ( mem_5, mem_6 );//Valor = 0 -> 'M'-'M' = 0
valor = strcmppgm2ram ( mem_1, mem_4 );//Valor = 0 -> 0 - 0 = 0
valor = strcmpram2pgm ( mem_6, mem_3 );//Valor = 0 -> 0 - 0 = 0
valor = strcmpram2pgm ( mem_5, mem_1 );//Valor = 10 -> 'M'-'C' = +10
}

}

```

4.19.4.9 **Strcpy, Strcpypgm, Strcpypgm2ram e Strcpyram2pgm**

A função **strcpy** copia a *string* apontada por *fonte* para a *string* apontada por *destino* e adiciona o caractere nulo de terminação. Essa função retorna o ponteiro do *destino*. Se *fonte* e *destino* forem sobrepostos, o comportamento dessa função é indefinido.

- **Função strcpypgm:** ambos os ponteiros apontam para a *string* localizada na memória de programa.
- **Função strcpypgm2ram:** o ponteiro *destino* aponta para a *string* localizada na memória de dados e o ponteiro *fonte* para a *string* localizada na memória de programa.
- **Função strcpyram2pgm:** o ponteiro *fonte* aponta para a *string* localizada na memória de dados e o ponteiro *destino* para a *string* localizada na memória de programa.

Sintaxe

```

ponteiro_d = strcpy ( destino, fonte )
ponteiro_d = strcpypgm ( destino, fonte )
ponteiro_d = strcpypgm2ram ( destino, fonte )
ponteiro_d = strcpyram2pgm ( destino, fonte )

```

Sendo:

- **destino:** ponteiro para *string*.
- **fonte:** ponteiro para *string*.
- **ponteiro_d:** ponteiro para o *destino*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata overlay minha_rom=0x400
    const rom char mem_3[ ]={"MPLAB C18."};//Armazena a string na memória de programa. [0x0400 - 0x040A]
#pragma romdata

void main( ) //Função principal
{
    char mem_1[21];//Armazena a string na memória de dados.
    const char mem_2[ ]={"Compilador "};//Armazena a string na memória de dados.
    strcpy ( mem_1, mem_2);//mem_1 = "Compilador "
    strcpypgm2ram ( &mem_1[11], mem_3 ); //mem_1 = "Compilador MPLAB C18."
}
```

4.19.4.10 Strcspn, Strcspnpgm, Strcspnpgmram e Strcspnrampgm

A função **strcspn** retorna o tamanho da *substring* formada pelos primeiros caracteres pertencentes à *string* apontada por *fonte*, que não estão na *string* apontada por *destino*. Caso nenhum caractere seja encontrado, a função retorna o tamanho da *string* apontada por *fonte*.

- **Função strcspnpgm:** ambos os ponteiros apontam para a *string* localizada na memória de dados.
- **Função strcspnpgmram:** o ponteiro *destino* aponta para a *string* localizada na memória de programa e o ponteiro *fonte* para o bloco da memória de dados.
- **Função strcspnrampgm:** o ponteiro *fonte* aponta para a *string* localizada na memória de programa e o ponteiro *destino* para o bloco da memória de dados.

Sintaxe

```
valor = strcspn ( destino, fonte )
valor = strcspnpgm ( destino, fonte )
valor = strcspnpgmram ( destino, fonte )
valor = strcspnrampgm ( destino, fonte )
```

Sendo:

- **destino:** ponteiro para *string*.
- **fonte:** ponteiro para *string*.
- **valor:** valor da *substring*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata minha_rom=0x400
    const rom char mem_2[ ]={"Versao v1.2"};//Armazena a string na memória de programa.
    const rom char mem_3[ ]="0123456789";//Armazena a string na memória de programa.
#pragma romdata

void main( ) //Função principal
{
    char mem_1[ ]="VERSAO V4.2";//Armazena a string na memória de dados.
    char mem_4[ ]=".2";//Armazena a string na memória de dados.
    char valor;

    valor = strcspn ( mem_1, mem_4);//valor = 9, o caractere da posição 9 da matriz mem_1 é ''
    valor = strcspnpgm ( mem_2, mem_3);//valor = 8, o caractere da posição 8 da matriz mem_2 é '1'
    valor = strcspnpgmram ( mem_2, mem_4 );//valor = 9, o caractere da posição 9 da matriz mem_2 é ''
    valor = strcspnrampgm ( mem_1, mem_3 );//valor = 8, o caractere da posição 8 da matriz mem_1 é '4'
    valor = strcspnpgmram ( mem_2, mem_4 );//valor = 9, o caractere da posição 9 da matriz mem_2 é ''
    valor = strcspnpgmram ( mem_3, mem_4 );//valor = 2, o caractere da posição 2 da matriz mem_3 é '2'
}
```

4.19.4.11 Strlen e Strlenpgm

A função **strlen** retorna o comprimento da *string* localizada na memória de dados, sem levar em consideração o caractere nulo de terminação.

A função **strlenpgm** é análoga à função da **strlen**, porém é destinada à *string* localizada na memória de programa.

Sintaxe

valor = strlen (string)
valor = strlenpgm (string)

Sendo:

- **string**: ponteiro para *string*.
- **valor**: comprimento da *string*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string
```

```
#pragma romdata overlay minha_rom=0x200
    const rom char mem_4[ ]="Versao v1.2"; //Armazena a string na memória de programa. [0x0200 - 0x020B]
    const rom char mem_5[ ]="0123456789"; //Armazena a string na memória de programa. [0x020C - 0x0216]
#pragma romdata

void main( ) //Função principal
{
    char mem_1[ ]="VERSAO V4.2"; //Armazena a string na memória de dados
    char mem_2[ ]=".2"; //Armazena a string na memória de dados.
    char mem_3[ ]={31,234,21,12,43,232,24,34,22,11,23,2,3,9,54}; //Armazena a string na memória de dados

    unsigned short int tamanho_ram;
    unsigned short long int tamanho_rom;

    tamanho_ram = strlen ( mem_1 ); //valor = 11.
    tamanho_ram = strlen ( mem_2 ); //valor = 2.
    tamanho_ram = strlen ( mem_3 ); //valor = 16.
    tamanho_rom = strlenpgm ( mem_4 ); //valor = 11.
    tamanho_rom = strlenpgm ( mem_5 ); //valor = 10.
}
```

4.19.4.12 Strlwr e Strlwrgm

A função **strlwr** converte em minúsculos, todos os caracteres presentes na *string* localizada na memória de dados, e **strlwrgm** na memória de programa.

Sintaxe

ponteiro = strlwr (string)
ponteiro = strlwrgm (string)

Sendo:

- **string**: ponteiro para *string*.
- **ponteiro**: ponteiro para *string*.

Exemplo

```
char mem_1[ ]="VERSAO V4.2"; //Armazena a string na memória de dados.
strlwr ( mem_1 ); //mem_1 = "versao v4.2"
```

4.19.4.13 Strncat, Strncatpgm, Strncatpgm2ram e Strncatram2pgm

A função **strncat** faz uma cópia de *n* caracteres da *string* apontada por *fonte* e a anexa no final da *string* apontada por *destino* e retorna o ponteiro do *destino*.

- **Função strncatpgm:** ambos os ponteiros apontam para a *string* localizada na memória de programa.
- **Função strncatpgm2ram:** o ponteiro *destino* aponta para a *string* localizada na memória de dados e o ponteiro *fonte* para a *string* localiza na memória de programa.
- **Função strncatram2pgm:** o ponteiro *fonte* aponta para a *string* localizada na memória de dados e o ponteiro *destino* para a *string* localiza na memória de programa.

Sintaxe

```
ponteiro_d = strncat ( destino, fonte, n )
ponteiro_d = strncatpgm ( destino, fonte, n )
ponteiro_d = strncatpgm2ram ( destino, fonte, n )
ponteiro_d = strncatram2pgm ( destino, fonte, n )
```

Sendo:

- **destino:** ponteiro para *string*.
- **fonte:** ponteiro para *string*.
- **ponteiro_d:** ponteiro para o *destino*.
- **n:** número de caracteres a serem copiados.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata overlay minha_rom=0x400
    rom char mem_3[11] = "VERSAO "; //Armazena a string na memória de programa. [0x0400 - 0x040A]
    rom char mem_4[ ] = "V1.2"; //Armazena a string na memória de programa. [0x040B - 0x040F]
#pragma romdata

#pragma idata access minha_ram=0x10
    char mem_1[26] = "SEJA BEM "; //Armazena a string na memória de dados. [0x0010 - 0x0029]
    char mem_2[ ] = "VINDO. "; //Armazena a string na memória de dados. [0x002A - 0x0030]
#pragma idata

void main( ) //Função principal
{
    char *ponteiro_ram;

    ponteiro_ram = strncat ( mem_1, mem_2, 6 ); //mem_1 = "SEJA BEM-VINDO." e ponteiro_ram = 0x0010
    //mem_1 = "SEJA BEM-VINDO. VERSAO " e ponteiro_ram = 0x0010
    ponteiro_ram = strncatpgm2ram ( mem_1, mem_3, 7 );
    //mem_1 = "SEJA BEM-VINDO. VERSAO V1.2" e ponteiro_ram = 0x0010
    ponteiro_ram = strncatpgm2ram ( mem_1, mem_4, 4 );
}
```

4.19.4.14 Strncmp, Strncmppgm, Strncmppgm2ram e Strncmpram2pgm

A função **strncmp** compara os *n* primeiros valores da *string* localizada na memória de dados apontada por *s_1*, com os valores da *string* localizada na memória de dados apontada por *s_2*. Essa função é um valor que se enquadra em uma das condições listadas em seguida.

- < 0: o primeiro byte que não combina em ambas as strings é menor no s_1 do que no s_2.
- = 0: conteúdo do s_1 é igual ao do s_2.
- > 0: o primeiro byte que não combina em ambas as strings é maior no s_1 do que no s_2.

A função **strncppgm** faz a comparação de duas strings localizadas na memória de programa.

A função **strncppgm2ram** faz a comparação de uma string localizada na memória de dados (s_1) e a outra na memória de programa (s_2).

A função **strncmpram2pgm** faz a comparação de uma string localizada na memória de dados (s_2) e a outra na memória de programa (s_1).

Sintaxe

```
relação_tamanho = strcmp ( s_1, s_2, n )
relação_tamanho = strncppgm ( s_1, s_2, n )
relação_tamanho = strncppgm2ram ( s_1, s_2, n )
relação_tamanho = strncmpram2pgm ( s_1, s_2, n )
```

Sendo:

- **s_1**: ponteiro para string.
- **s_2**: ponteiro para string.
- **relação_tamanho**: valor do tipo **signed char**.
- **n**: número de caracteres a serem comparados.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string

#pragma romdata overlay minha_rom=0x4f0
    rom char mem_3[ ]="MEMORIA DE PROGRAMA";//Armazena a string na memória de programa.
    rom char mem_4[ ]="MEMoria DE PROGRAMA";//Armazena a string na memória de programa.
#pragma romdata

void main( ) //Função principal
{
    char mem_1[]="MEMORIA DE DADOS "//Armazena a string na memória de dados.
    char mem_2[]="MEMORIA de DADOS";//Armazena a string na memória de dados.

    char mem_5[]={33,22,45,2,45,8,65,44,3,78};//Armazena a string na memória de dados.
    char mem_6[]={33,22,45,2,45,0,65,44,3,78};//Armazena a string na memória de dados.

    signed char valor;

    valor = strcmp ( mem_1, mem_2, 7 );//valor = 0 -> 'A'-'A'=0
    valor = strcmp ( mem_1, mem_2, sizeof(mem_1) );//valor = -32 -> 'D'-'d'=-32
    valor = strcmp ( mem_5, mem_6, sizeof(mem_1) );//valor = 8 -> 8-0=8
    valor = strcmp ( mem_6, mem_5, sizeof(mem_6) );//valor = -8 -> 0-8=-8
    valor = strncppgm2ram ( mem_1, mem_3, 8 );//valor = 0 -> ' '-' ' =0
    valor = strncppgm2ram ( mem_1, mem_3, sizeof(mem_2) );//valor = -12 -> 'D'-'P'=-12
    valor = strncmpram2pgm ( mem_4, mem_1, 10 );//valor = 32 -> 'e'-'E'=32
    valor = strncmpram2pgm ( mem_4, mem_2, sizeof(mem_4) );//valor = 32 -> 'e'-'E'=32
}
```

4.19.4.15 Strncpy, Strncpypgm, Strncpypgm2ram e Strncpyram2pgm

A função **strncpy** copia os *n* primeiros caracteres da *string* apontada por *fonte* para a *string* apontada por *destino*. Se o número de caracteres *n* for maior que a matriz, serão copiados para o *destino* somente os caracteres que antecedem o caractere nulo de terminação. Após realizar a tarefa, a função retorna o ponteiro do *destino*. Se *fonte* e *destino* forem sobrepostos, o comportamento dessa função é indefinido.

- **Função strncpypgm:** ambos os ponteiros apontam para a *string* localizada na memória de programa.
- **Função strncpypgm2ram:** o ponteiro *destino* aponta para a *string* localizada na memória de dados e o ponteiro *fonte* para o bloco da memória de programa.
- **Função strncpyram2pgm:** o ponteiro *fonte* aponta para a *string* localizada na memória de dados e o ponteiro *destino* para o bloco da memória de programa.

Sintaxe

```
ponteiro_d = strncpy ( destino, fonte, n )
ponteiro_d = strncpypgm ( destino, fonte, n )
ponteiro_d = strncpypgm2ram ( destino, fonte, n )
ponteiro_d = strncpyram2pgm ( destino, fonte, n )
```

Sendo:

- **destino:** ponteiro para *string*.
- **fonte:** ponteiro para *string*.
- **ponteiro_d:** ponteiro para o *destino*.
- **n:** número de caracteres a serem copiados.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata overlay minha_rom=0x4f0
    rom char mem_3[ ]="PIC18.";//Armazena a string na memória de programa. [0x04F0 - 0x04F6]
#pragma romdata

#pragma idata access minha_ram=0x00
    char mem_1[36]="PROGRAMANDO ";//Armazena a string na memória de dados. [0x0000 - 0x0023]
    char mem_2[ ]="MICROCONTROLADORES ";//Armazena a string na memória de dados. [0x0024 - 0x0037]
#pragma idata

void main( ) //Função principal
{
    //mem_1="PROGRAMANDO MICROCONTROLADORES "
    strncpy ( &mem_1[12], mem_2, sizeof(mem_2) );
    //mem_1="PROGRAMANDO MICROCONTROLADORES PIC"
    strncpypgm2ram ( &mem_1[31], mem_3, 3 );
    //mem_1="PROGRAMANDO MICROCONTROLADORES PIC18"
    strncpypgm2ram ( &mem_1[34], &mem_3[3], 2 );
}
```

4.19.4.16 Strpbrk, Strpbrkpgm, Strpbrkpgmram e Strpbrkrampgm

A função **strpbrk** retorna um ponteiro para a primeira ocorrência de um dos caracteres pertencentes à *fonte* no *destino*. Caso não seja verificada nenhuma ocorrência de *c*, a função retorna um ponteiro nulo.

- **Função strpbrkpgm:** ambos os ponteiros apontam para a *string* localizada na memória de dados
- **Função strpbrkpgmram:** o ponteiro *destino* aponta para a *string* localizada na memória de programa e o ponteiro *fonte* para a *string* localizada na memória de dados.
- **Função strpbrkrampgm:** o ponteiro *fonte* aponta para a *string* localizada na memória de programa e o ponteiro *destino* para a *string* localizada na memória de dados.

Sintaxe

```
ponteiro_d = strpbrk ( destino, fonte )
ponteiro_d = strpbrkpgm ( destino, fonte )
ponteiro_d = strpbrkpgmram ( destino, fonte )
ponteiro_d = strpbrkrampgm ( destino, fonte )
```

Sendo:

- **destino:** ponteiro para *string*.
- **fonte:** ponteiro para *string*.
- **ponteiro_d:** ponteiro para o *destino*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata overlay minha_rom=0x600
    rom char mem_3[ ]="PIC18";//Armazena a string na memória de programa. [0x0600 - 0x0605]
    rom char mem_4[ ]="TESTE 8.0";//Armazena a string na memória de programa. [0x0606 - 0x060F]
#pragma romdata
#pragma idata access minha_ram=0x00
    char mem_1[ ]="0123456789";//Armazena a string na memória de dados. [0x00 - 0x0A]
    char mem_2[ ]="Teste 8";//Armazena a string na memória de dados. [0x0B - 0x12]
#pragma idata
void main( ) //Função principal
{
    char *ponteiro_ram;
    rom char *ponteiro_rom;
    ponteiro_ram = strpbrk ( mem_2, mem_1 ), //valor=0x0011;
    ponteiro_rom = strpbrkpgm ( mem_4, mem_3 ); //valor=0x060C
    ponteiro_rom = strpbrkpgmram ( mem_4, mem_2 );//valor=0x0606
    ponteiro_ram = strpbrkrampgm ( mem_1, mem_3 );//valor=0x0001
}
```

4.19.4.17 Strrchr

A função **strrchr** localiza a última ocorrência de *c* (**unsigned char**) dentro da *string* localizada na memória de dados apontada por *s* e retorna um ponteiro para o caractere dentro da *string*. Se não for verificada nenhuma ocorrência de *c*, a função retorna um ponteiro nulo.

Sintaxe

```
localização = strrchr ( s, c )
```

Sendo:

- **s:** ponteiro para *string*.
- **c:** valor do dado **unsigned char**.
- **localização:** ponteiro para o dado *c* dentro da *string*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.

#pragma idata access minha_ram=0x00
    char mem_1[ ]="0123456789"; //Armazena a string na memória de dados. [0x00 - 0x0A]
    char mem_2[ ]="Teste 8"; //Armazena a string na memória de dados. [0x0B - 0x12]
#pragma idata

void main( ) //Função principal
{
    char *ponteiro_ram;
    ponteiro_ram = strchr( mem_1, '4' ); //ponteiro_ram=0x0004;
    ponteiro_ram = strchr( mem_2, 'l' ); //ponteiro_ram=0x000E;
}
```

4.19.4.18 Strspn, Strspnpgm, Strspnpgmram e Strspnrampgm

A função **strspn** retorna o número de caracteres consecutivos da string apontada por *s_1*, os quais estão na string apontada por *s_2*.

- » **Função strspnpgm:** ambos os ponteiros apontam para a *string* localizada na memória de dados.
- » **Função strspnpgmram:** o ponteiro *s_1* aponta para a *string* localizada na memória de programa e o ponteiro *s_2* para a *string* localizada na memória de dados.
- » **Função strspnrampgm:** o ponteiro *s_2* aponta para a *string* localizada na memória de programa e o ponteiro *s_1* para a *string* localizada na memória de dados.

Sintaxe

```
localização = strspn ( s_1, s_2 )
localização = strspnpgm ( s_1, s_2 )
localização = strspnpgmram ( s_1, s_2 )
localização = strspnrampgm ( s_1, s_2 )
```

Sendo:

- ***s_1*:** ponteiro para *string*.
- ***s_2*:** ponteiro para *string*.
- **localização:** ponteiro para o dado *c* dentro da *string*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata overlay minha_rom=0x600
    rom char mem_5[ ]="banana"; //Armazena a string na memória de programa. [0x0600 - 0x0606]
    rom char mem_6[ ]="ba"; //Armazena a string na memória de programa. [0x0607 - 0x0609]
    rom char mem_7[ ]="an"; //Armazena a string na memória de programa. [0x060A - 0x060C]
    rom char mem_8[ ]="banp"; //Armazena a string na memória de programa. [0x060D - 0x0610]
#pragma romdata

#pragma idata access minha_ram=0x00
    char mem_1[ ]="papagaio"; //Armazena a string na memória de dados. [0x00 - 0x08]
    char mem_2[ ]="pa"; //Armazena a string na memória de dados. [0x09 - 0x0B]
    char mem_3[ ]="iapb"; //Armazena a string na memória de dados. [0x0C - 0x0F]
```

```

char mem_4[ ]="iao";//Armazena a string na memória de dados.      [0x10 - 0x13]
#pragma idata

void main( ) //Função principal
{
    unsigned short int valor,
    valor = strspn ( mem_1, mem_2 );//valor=4;
    valor = strspn ( mem_1, mem_3 );//valor=4;
    valor = strspn ( mem_1, mem_4 );//valor=0;
    valor = strspnpgm( mem_5, mem_6 );//valor=2;
    valor = strspnpgm( mem_5, mem_7 );//valor=0;
    valor = strspnpgm( mem_5, mem_8 );//valor=6;
    valor = strspnpgmram( mem_5, mem_3 );//valor=2;
    valor = strspnrampgm( mem_1, mem_8 );//valor=4;
}

```

4.19.4.19 Strstr, Strstrpgm, Strstrpgmram e Strstrrampgm

A função **strstr** retorna a primeira ocorrência da *string* apontada por *s_2* dentro da *string* apontada por *s_1* e retorna um ponteiro para *s_1*; caso contrário, retorna um ponteiro nulo.

- **Função strstrpgm:** ambos os ponteiros apontam para a *string* localizada na memória de dados.
- **Função strstrpgmram:** o ponteiro *s_2* aponta para a *string* localizada na memória de programa e o ponteiro *s_1* para o bloco da memória de dados.
- **Função strstrrampgm:** o ponteiro *s_1* aponta para a *string* localizada na memória de programa e o ponteiro *s_2* para o bloco da memória de dados.

Sintaxe

```

ponteiro_s_1 = strstr ( s_1, s_2 )
ponteiro_s_1= strstrpgm ( s_1, s_2 )
ponteiro_s_1= strstrpgmram ( s_1, s_2 )
ponteiro_s_1= strstrrampgm ( s_1, s_2 )

```

Sendo:

- ***s_1*:** ponteiro para *string*.
- ***s_2*:** ponteiro para *string*.
- **ponteiro_s_1:** ponteiro para *string* apontada por *s_1*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.
```

```
#pragma romdata overlay minha_rom=0x600
rom char mem_5[ ]="paparrazzo";//Armazena a string na memória de programa. [0x0600 - 0x0609]
rom char mem_6[ ]="pa";//Armazena a string na memória de programa.      [0x060A - 0x060C]
rom char mem_7[ ]="ra";//Armazena a string na memória de programa.      [0x060D - 0x060F]
rom char mem_8[ ]="para";//Armazena a string na memória de programa.     [0x0610 - 0x0614]
#pragma romdata
```

```
#pragma idata access minha_ram=0x00
char mem_1[ ]="papagaio";//Armazena a string na memória de dados. [0x00 - 0x08]
char mem_2[ ]="pa";//Armazena a string na memória de dados.      [0x09 - 0x0B]
char mem_3[ ]="gai";//Armazena a string na memória de dados.      [0x0C - 0x0F]
```

```

char mem_4[ ]="pag";//Armazena a string na memória de dados.      [0x10 - 0x13]
#pragma idata

void main( ) //Função principal
{
    char *ponteiro_ram;
    rom char *ponteiro_rom;

    ponteiro_ram = strstr ( mem_1, mem_2 );//valor=0x0000
    ponteiro_ram = strstr ( mem_1, mem_3 );//valor=0x0004
    ponteiro_ram = strstr ( mem_1, mem_4 );//valor=0x0002
    ponteiro_rom = strstrupgm( mem_5, mem_6 );//valor=0x0600
    ponteiro_rom = strstrupgm( mem_5, mem_7 );//valor=0x0604
    ponteiro_rom = strstrupgmrram( mem_5, mem_3 );//valor=0x0000
    ponteiro_ram = strstrrampgm( mem_1, mem_6 );//valor=0x0000
}

```

4.19.4.20 **Strtok, Strtokpgm, Strtokpgmram e Strtokrampgm**

A função retorna um ponteiro para o próximo conjunto de caracteres da *string* e que estão delimitados por um dos caracteres do *delimitador*.

Sintaxe

```

ponteiro = strtok ( string, delimitador )
ponteiro = strtokpgm ( string, delimitador )
ponteiro = strtokpgmram ( string, delimitador )
ponteiro = strtokrampgm ( string, delimitador )

```

Sendo:

- **string:** ponteiro para *string*.
- **delimitador:** ponteiro para *string*.
- **ponteiro:** ponteiro para *string*.

A função começa buscando em *string* a ocorrência do primeiro caractere que não está no *delimitador*, e retorna um ponteiro nulo, caso não for encontrado. Se o caractere for encontrado, a função retorna um ponteiro para sua posição, definindo o início da seção que vai se estender até o final da *string* ou até que um caractere presente no *delimitador* seja encontrado. A pesquisa subsequente é iniciada a partir da última posição apontada.

Se um dos caracteres apontados pelo *delimitador* for encontrado em *string*, logo ele é substituído por um caractere nulo.



Cada chamada subsequente deve adicionar a constante zero como primeiro argumento da função, para que a pesquisa seja iniciada a partir da posição apontada.

Exemplo

```

#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.

```

```

#pragma romdata overlay minha_rom=0x600
rom char mem_3[ ]="CompiladorxMPLAB/C18"; //Armazena a string na memória de programa. [0x0600 - 0x0614]

```

```

    rom char mem_4[ ]="x"]; //Armazena a string na memória de programa.[0x0615 - 0x0618]
#pragma romdata

#pragma idata access minha_ram=0x00
    char mem_1[ ]="COMO|PROGRAMAR|O|PIC9OK"; //Armazena a string na memória de dados [0x00 - 0x17]
    char mem_2[ ]="9"; //Armazena a string na memória de dados [0x18 - 0x1A]
#pragma idata

void main( ) //Função principal
{
    char *ponteiro_ram;
    rom char *ponteiro_rom;

    ponteiro_ram = strtok(mem_1,mem_2); //ponteiro_ram = 0x0000 e string = "COMO.PROGRAMAR|O|PIC9OK"
    ponteiro_ram = strtok(0,mem_2); //ponteiro_ram = 0x0005 e string = "COMO.PROGRAMAR.O|PIC9OK"
    ponteiro_ram = strtok(0,mem_2); //ponteiro_ram = 0x00f e string = "COMO.PROGRAMAR.O.PIC9OK"
    ponteiro_ram = strtok(0,mem_2); //ponteiro_ram = 0x011 e string = "COMO.PROGRAMAR.O.PIC.OK"
    ponteiro_rom = strtokpgm(mem_3,mem_4); //ponteiro_rom = 0x0600 e string = "CompiladorxMPLAB/C18"
    ponteiro_rom = strtokpgm(0,mem_4); //ponteiro_rom = 0x060B e string = "CompiladorxMPLAB/C18"
    ponteiro_rom = strtokpgm(0,mem_4); //ponteiro_rom = 0x0611 e string = "CompiladorxMPLAB/C18"
}

```

4.19.4.21 Strupr e Struprpgm

A função **strupr** converte em maiúsculos, todos os caracteres presentes na *string* localizada na memória de dados, e **struprpgm** na memória de programa.

Sintaxe

```

ponteiro = strupr ( string )
ponteiro = struprpgm ( string )

```

Sendo:

- **string**: ponteiro para *string*.
- **ponteiro**: ponteiro para *string*.

Exemplo

```

#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <string.h> //Adiciona a biblioteca de funções de manipulação de string.

```

```

#pragma romdata overlay minha_rom=0x00
    rom char mem_2[ ]="paparazzo"; //Armazena a string na memória de programa.[0x0600 - 0x0609]
#pragma romdata
#pragma idata access minha_ram=0x00
    char mem_1[ ]="papagaio"; //Armazena a string na memória de dados. [0x00 - 0x08]
#pragma idata
void main( ) //Função principal
{
    char *ponteiro_ram;
    rom char *ponteiro_rom;
    ponteiro_ram = strupr ( mem_1 );//mem_1="PAPAGAO"
}

```

4.19.5 Funções Matemáticas

As funções matemáticas da linguagem C padrão estão declaradas no arquivo `math.h`. A utilização das funções listadas neste tópico requer a introdução da diretiva `#include <math.h>` dentro do código de programa, conforme o modelo apresentado pela função `acos`.

As operações aritméticas de ponto flutuante realizadas pelo compilador MPLAB® C18 estão de acordo com o padrão IEEE 754 tanto no modo tradicional quanto no modo estendido. O número de ponto flutuante é representado por um bit de sinal (*s*), expoente (*e*) e os dígitos da mantissa na base binária (*d*), como pode ser observado na tabela a seguir.

Tabela 4.17: Representação do ponto flutuante.

Sinal (<i>s</i>)	Expoente (<i>e</i>)	Mantissa (<i>d</i>)
±	e ₇ e ₆ e ₅ e ₄ e ₃ e ₂ e ₁ e ₀	d ₂₃ d ₂₂ d ₂₁ ...d ₃

O padrão IEEE 754 também recomenda o número de bits para representação do número, de acordo com a precisão adotada. Por exemplo, o MPLAB® C18 adota o tamanho de 32bits; para este tamanho são recomendados: *s* = 1, *e* = 8 e *d* = 23.

Esses componentes estão na forma:

$$x = (-1)^s \cdot d_0.d_1.d_2.d_3....d_{23} \cdot 2^e$$

Tabela 4.18: Bits de representação do ponto flutuante.

	eb	f0	f1	f2
IEEE-754 32-bit	seee eeee	eddd dddd	dddd dddd	dddd dddd
Microchip 32-bit	eeee eeee	sddd dddd	dddd dddd	ddd dddd

Legenda: *s* - bit de sinal, *e* - expoente; *d* - dígitos da mantissa na base binária.

O compilador MPLAB® C18 permite representar os números de ponto flutuante na forma adotada pela Microchip, de acordo com a Tabela 4.18.

Todos os ângulos de entrada e saída das funções são expressos em radianos. Para convertê-los em graus, ou vice-versa, pode ser utilizada uma das seguintes equações:

$$\text{Valor_graus} = \frac{180 * \text{Valor_rad}}{\pi} [{}^\circ] \quad \text{Valor_rad} = \frac{\pi * \text{Valor_graus}}{180} [\text{rad}]$$



As funções matemáticas listadas neste tópico retornam um `NaN` (*Not a Number*) se os argumentos de entradas forem inválidos.

4.19.5.1 Acos, Asin, Atan e Atan2

- **acos:** calcula o arco cosseno do operando (argumento) compreendido entre -1 e $+1$ e retorna o ângulo expresso em radianos (0 a π).
- **asin:** calcula o arco seno do operando (argumento) compreendido entre -1 e $+1$ e retorna o ângulo expresso em radianos ($-\pi/2$ a $+\pi/2$).
- **atan:** calcula o arco tangente do operando (argumento) e retorna o ângulo expresso em radianos ($-\pi/2$ a $+\pi/2$).
- **atan2:** calcula o arco tangente de x/y e retorna o ângulo expresso em radianos ($-\pi$ a $+\pi$).

Sintaxe

```
ângulo = acos ( operando )
ângulo = asin ( operando )
ângulo = atan ( operando )
ângulo = atan2 ( x , y )
```

Sendo:

- **operando, x e y:** valor do tipo float.
- **ângulo:** valor do tipo float.

Exemplo

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <math.h> //Adiciona a biblioteca de funções matemáticas
```

```
void main (void)
{
    float angulo_rad; //Declara uma variável do tipo float

    angulo_rad = asin (1);           // angulo_rad = 1.570796 -> pi/2.
    angulo_rad = asin (0);          // angulo_rad = 0.
    angulo_rad = asin (4);          // angulo_rad = NaN
    angulo_rad = acos (0);          // angulo_rad = 1.570796 -> pi/2
    angulo_rad = acos (-1);         // angulo_rada = 3.141593 -> pi.
    angulo_rad = acos (0.5);        // angulo_rada = 1.047198 -> pi/3.
    angulo_rad = atan (1);          // angulo_rad = 0.7853982 -> pi/4.
    angulo_rad = atan2 (0,0);       // angulo_rad = NaN.
    angulo_rad = atan2 (20.0,2.0);  // angulo_rad = 1.471128 -> 0,4682744826pi
}
```

4.19.5.2 Ceil e Floor

- **ceil:** retorna um *valor* inteiro correspondente ao arredondamento para baixo de um *número* fracionário.
- **floor:** retorna um *valor* inteiro correspondente ao arredondamento para cima de um *número* fracionário.

Sintaxe

```
valor = ceil ( número )
valor = floor ( número )
```

Sendo:

- **número:** valor do tipo float.
- **valor:** valor do tipo float.

Exemplo

```
float valor_int; //Declara uma variável do tipo float.
```

```
valor_int = ceil(-2.83);    //valor_int = -2
valor_int = floor (-2.83);  //valor_int = -3
valor_int = ceil (2.40);    //valor_int = 3
valor_int = floor (2.40);   //valor_int = 2
```

4.19.5.3 Cos, Sin e Tan

As funções **cos**, **sin** e **tan** retornam o valor do cosseno, seno e tangente do ângulo expresso em radianos.

Sintaxe

```
valor = cos ( ângulo )
valor = sin ( ângulo )
valor = tan ( ângulo )
```

Sendo:

- **valor**: valor do tipo **float**.
- **ângulo**: ângulo expresso em radianos.

Exemplo

```
#define pi 3.141592654
```

```
void main( ) //Função principal
{
    float valor_saida; //Declara uma variável do tipo float.

    valor_saida = sin (pi/2); // valor_saida = 1.
    valor_saida = sin (pi/4); // valor_saida = 0.7071068.
    valor_saida = cos (pi); // valor_saida = -1.
    valor_saida = cos (2*pi); // valor_saida = 1.
    valor_saida = tan (pi/4); // valor_saida = 1.
}
```

4.19.5.4 Cosh, Sinh e Tanh

As funções **cosh**, **sinh** e **tanh** retornam o valor do cosseno, seno e tangente hiperbólica do argumento de entrada (**valor_entrada**).

$$\cosh(x) = \frac{e^x + e^{-x}}{2} \quad \sinh(x) = \frac{e^x - e^{-x}}{2} \quad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Sintaxe

```
valor = cosh (valor_entrada )
valor = sinh (valor_entrada )
valor = tanh (valor_entrada )
```

Sendo:

- **valor**: valor do tipo **float**.
- **valor_entrada**: valor do tipo **float**.

Exemplo

```
float valor_saida; //Declara uma variável do tipo float.
```

```
valor_saida = sinh (2); // valor_saida = 3.626860.
valor_saida = cosh (5.3); // valor_saida = 100.1709.
valor_saida = tanh (1.8); // valor_saida = 0.9468060.
```

4.19.5.5 Exp

A função **exp** retorna o logaritmo natural e elevado à potência *pot*.

$$\text{valor} = e^{\text{pot}}$$

Sintaxe

valor = **exp** (*pot*)

Sendo:

- **valor**: valor do tipo **float**.
- **pot**: valor do tipo **float**.

Exemplo

float valor; //Declara uma variável do tipo float.

```
valor = exp (0.5);           //valor = 1.648721.  
valor = exp (1);            //valor = 2.718282
```

4.19.5.6 Fabs

Calcula o valor absoluto de um *número*.

Sintaxe

valor = **fabs** (*número*)

Sendo:

- **valor**: valor do tipo **float**.
- **número**: valor do tipo **float**.

Exemplo

float valor; //Declara uma variável do tipo float.

```
valor = fabs (-3421.3234);          //valor_float = 3421.323.  
valor = fabs (-9482);                //valor_int16 = 9482.  
valor = fabs (4039);                 //valor_int16 = 4039.  
valor = fabs (-303.22);              //valor_float = 303.22.
```

4.19.5.7 Fmod

A função **fmod** retorna o resto da divisão entre dois números fracionários.

$$\text{resposta} = \frac{\text{valor_1}}{\text{valor_2}}$$

Sintaxe

resposta = **fmod** (*valor_1*, *valor_2*)

Sendo:

- **resposta**: valor do tipo **float**.
- **valor_1** : valor do tipo **float**.
- **valor_2** : valor do tipo **float**.

Exemplo

```
float valor_1, valor_2, valor_3, valor_4, resposta; //Declara cinco variáveis do tipo float.

valor_1 = 21; //Atribui o valor decimal 21 a variável valor_1.
valor_2 = 2; //Atribui o valor decimal 2 a variável valor_2.
valor_3 = 85.43; //Atribui o valor decimal 85.43 a variável valor_3.
valor_4 = 23.37; //Atribui o valor decimal 23.37 a variável valor_4

resposta = fmod (valor_1, valor_2); //resposta = 1.
resposta = fmod (valor_3, valor_4); //resposta = 15.32
```

4.19.5.8 Frexp

Divide o argumento *valor* em duas partes de acordo com a equação apresentada a seguir:

$$\text{valor} = \text{parte_fracionária} * 2^{\text{pexp}}$$

A função escolhe o valor de *pexp* de modo que a *parte_fracionária* esteja entre 0,5 e 1.

Sintaxe

parte_fracionária = **frexp** (*valor*, *pexp*)

Sendo:

- **parte_fracionária**: valor do tipo **float**.
- **valor**: valor do tipo **float**.
- **pexp**: endereço para uma variável do tipo **int**.

Exemplo

```
float parte_f;
float valor_1 = 8, valor_2 = 12, valor_3 = -84.33, valor_4 = 921.987;
int pexp;
```

```
parte_f = frexp( valor_1, &pexp ); //parte_f = 0.5582500 e pexp = 4 -> valor = 0.5*2^4=8
parte_f = frexp( valor_2, &pexp ); //parte_f = 0.7500000 e pexp = 4 -> valor = 0.75*2^4=12
parte_f = frexp( valor_3, &pexp ); //parte_f = -0.6588281 e pexp = 7 -> valor = -0.6588281*2^7=-84.3299968
parte_f = frexp( valor_4, &pexp ); //parte_f = 0.9003779 e pexp = 10 -> valor = 0.9003779*2^10=921.9869696
```

4.19.5.9 IeeeTomchp e Mchptoieee

A função **ieeeTomchp** converte um *valor* de ponto flutuante de 32bits no formato IEEE-754 para o formato MICROCHIP, enquanto a função **mchptoieee** faz o inverso.

Tabela 4.19: Representação do ponto flutuante.

	eb	f0	f1	f2
IEEE-754 32-bit	seee eeee	eddd dddd	dddd dddd	dddd dddd
Microchip 32-bit	eeee eeee	sddd dddd	dddd dddd	dddd dddd

Legenda: s - bit de sinal, e - expoente; d - dígitos da mantissa na base binária.

Sintaxe

```
f_microchip = ieeeTomchp ( valor_ieee )
f_ieee = mchptoieee ( valor_microchip )
```

Sendo:

- **f_microchip**: valor do tipo **long**.
- **f_ieee**: valor do tipo **float**.
- **valor_ieee**: valor do tipo **float**.
- **valor_microchip**: valor do tipo **long**.

Exemplo

```
float f_ieee; //Declara uma variável do tipo float.  
long f_microchip; //Declara uma variável do tipo long.  
f_microchip = ieee2tomchp ( 8.22 ); //f_microchip ≈ -2113698529  
f_ieee = mchptoiieee ( -2113698529 ); //f_ieee = 8.220000
```

4.19.5.10 Ldexp

Calcula a equação a seguir:

$$\text{resposta} = \text{valor} * 2^{\text{exp}}$$

Sintaxe

`resposta = ldexp (valor, exp)`

Sendo:

- **resposta**: valor do tipo **float**.
- **valor**: valor do tipo **float**.
- **exp**: valor do tipo **int**.

Exemplo

```
float resposta;
```

```
resposta = ldexp( 2.2, 3 ); //resposta = 2.2*2^3 = 17.60000.  
resposta = ldexp( 9, 1 ); //resposta = 9*2^1 = 18.00000.  
resposta = ldexp( -5.5, 5 ); //resposta = -5.5*2^5 = -176.0000.  
resposta = ldexp( -10, 10 ); //resposta = -10*2^10 = -10240.00.
```

4.19.5.11 Log e Log10

A função **log** retorna o logaritmo natural (**e**) de um *número*, enquanto a função **log10** devolve o logaritmo de base 10 de um *número*.

O valor atribuído a **e** é igual a 2.718282.

Sintaxe

`valor = log (número)`
`valor = log10 (número)`

Sendo:

- **valor**: valor do tipo **float**.
- **número**: valor do tipo **float**.

A função **log** pode ser interpretada como: $\log_e \text{número} = \text{valor}$.

A função **log10** pode ser interpretada como: $\log_{10} \text{número} = \text{valor}$.

Exemplo

```
float valor; //Declara uma variável do tipo float.  
  
valor = log (exp(1)); //valor = 1.  
valor = log (3.48); //valor = 1.247032.  
valor = log10 (100); //valor = 2.  
valor = log10 (300); //valor = 2.477121
```

4.19.5.12 Modf

A função **modf** decompõe o *número* em seu valor inteiro e fracionário. Ela retorna o valor fracionário e inserre o valor inteiro no endereço da variável *valor_int*.

Sintaxe

```
resposta = modf ( número, valor_int )
```

Sendo:

- **resposta**: valor do tipo **float**.
- **número**: valor do tipo **float**.
- **valor_int**: endereço para uma variável do tipo **float**.

Exemplo

```
float valor_1, valor_2, resposta, valor_int; //Declara quatro variáveis do tipo float  
  
valor_1 = 21.42; //Atribui o valor 21.42 à variável valor_1.  
valor_2 = 4829.8372; //Atribui o valor 4829.8372 à variável valor_2.  
  
resposta = modf (valor_1,&valor_int); //resposta = 0.42 e valor_int = 21.  
resposta = modf (valor_2,&valor_int); //resposta = 0.8374023 e valor_int = 4829.
```

4.19.5.13 Pow

A função **pow** retorna o valor da *base* elevada à *potência*.

$$\text{resposta} = \text{base}^{\text{potência}}$$
Sintaxe

```
resposta = pow (base, potência )
```

Sendo:

- **resposta**: valor do tipo **float**.
- **base**: valor do tipo **float**.
- **potência**: valor do tipo **float**.

Exemplo

```
float valor_1, valor_2, resposta; //Declara três variáveis do tipo float.  
  
valor_1 = 2; //Atribui o valor 2 à variável valor_1.  
valor_2 = 3; //Atribui o valor 3 à variável valor_2.  
resposta = pow (valor_1,valor_2); //resposta = 2^3 = 8.  
resposta = pow (valor_2,valor_1); //resposta = 3^2 = 9.  
resposta = pow (9,0.5); //resposta = 9^0,5 = 3.  
resposta = pow (valor_1,4); //resposta = 2^4 = 16.
```

4.19.5.14 Sqrt

A função **sqrt** retorna o valor correspondente à raiz quadrada de um *número*.

$$\text{resposta} = \sqrt{\text{número}}$$

Sintaxe

resposta = **sqrt** (*número*)

Sendo:

- **resposta**: valor do tipo **float**.
- **número**: valor do tipo **float**.

Exemplo

```
float valor_1, valor_2, resposta; //Declara três variáveis do tipo float.  
valor_1 = 9; //Atribui o valor 9 à variável valor_1.  
valor_2 = 64; //Atribui o valor 64 à variável valor_2.
```

```
resposta = sqrt (valor_1); //resposta = 3.  
resposta = sqrt (valor_2); //resposta = 8.  
resposta = sqrt (-32); //resposta = NaN.  
resposta = sqrt (12); //resposta = 3.464102.
```

4.19.6 Números Pseudoaleatórios

As duas funções relacionadas à geração de números pseudoaleatórios são **rand** e **srand**. A utilização das funções listadas neste tópico requer a introdução da diretiva `#include < stdlib.h >` dentro do código de programa, conforme o modelo apresentado pela função **rand**.

4.19.6.1 Rand

A função **rand** retorna um número pseudoaleatório, compreendido entre 0 e 32.767.

Sintaxe

valor = **rand**()

Sendo:

- **valor**: valor de 16bits retornado pela função.

Exemplo

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.  
#include <stdlib.h> //Adiciona a biblioteca de funções miscelâneas.
```

```
void main (void)  
{  
    unsigned int matriz[10];  
    unsigned char cont;  
  
    for(cont=0 ; cont<10 ; cont++)  
    {  
        matriz[cont] = rand( ); //Armazena números pseudoaleatórios na matriz.  
    }  
}
```

4.19.6.2 Srand

A função **srand** define um ponto de partida para uma nova sequência de números pseudoaleatórios, que serão retornados por sucessivas chamadas da função **rand**. O gerador pode ser reiniciado se o *valor* for igual a 1.

Sintaxe

srand(valor)

Sendo:

- **valor:** valor de 32bits.



Se a função **srand** for chamada com o mesmo *valor* de ponto de partida, os números pseudoaleatórios gerados podem coincidir com os anteriores. Caso a função **rand** seja chamada antes de qualquer **srand**, então os números pseudoaleatórios podem coincidir com os números gerados com o *valor* de ponto de partida igual a 1.

Exemplo

```
unsigned int matriz1[10], matriz2[10];
unsigned char cont;

for(cont=0 ; cont<10 ; cont++)
{
    srand(rand()); //Define um novo ponto de partida a cada iteração.
    matriz1[cont] = rand(); //Armazena os números pseudoaleatórios na matriz.
}

srand(92332); //Define um ponto de partida para uma nova sequência de números pseudoaleatórios
for(cont=0 ; cont<10 ; cont++)
{
    matriz2[cont] = rand(); //Armazena os números pseudoaleatórios na matriz.
}
```

4.20 Código em Assembly

Assembly é uma linguagem de baixo nível, que lida diretamente com a linguagem da máquina. Com ela é possível criar códigos mais eficientes principalmente quando se trata de rotina de interrupção ou alguma ação que deve ser tomada quase que imediatamente. A linguagem Assembly possibilita criar códigos mais enxutos, rápidos e eficientes, no entanto é bastante trabalhoso.

O MPLAB® C18 permite introduzir códigos Assembly em um código escrito em C, permitindo ao programador criar códigos mais eficientes em determinadas partes do programa. A inserção de código Assembly deve ser feita de acordo com a sintaxe a seguir.

Sintaxe

_asm

instrução argumentos

_endasm

Sendo:

- **instrução:** instruções Assembly. Veja as Tabelas 4.20 e 4.21.
- **argumentos:** argumentos da instrução separados por uma vírgula ','.

A seguir estão listados todos os elementos utilizados pelas instruções, bem como as convenções utilizadas por elas.

Tabela 4.20: Descrição dos componentes das instruções em Assembly.

Campo	Descrição
Arquivos de registro	
Destino	Destinação para o registrador WREG ou SRF (Specified Register File).
f	<p>Endereço do arquivo de registro.</p> <p>f - 8bits (00h to FFh) ou designa 2bits para o FSR (0h a 3h).</p> <p>f - 12bits (000h to FFFFh). É o endereço da fonte.</p> <p>f' - 12bits (000h to FFFFh). É o endereço do destino.</p>
r	Número do FSR. 0, 1 ou 2.
x	Não precisa se preocupar com o valor de x, pois o compilador Assembly gera o código com x=0.
z	<p>Endereçamento indireto deslocado.</p> <p>z' - 7bits. Valor do offset para o endereçamento indireto do arquivo de registro. (fonte)</p> <p>z'' - 7bits. Valor do offset para o endereçamento indireto do arquivo de registro. (destino)</p>
Literal	
k	<p>Constante, campo ou rótulo literal.</p> <p>k - 4bits.</p> <p>kk - 8bits.</p> <p>kkk - 12bits.</p>
Offset, incremento/decremento	
n	<p>Endereço relativo.</p> <p>*</p> <p>*+</p> <p>*-</p> <p>+*</p> <p>Comportamento do registrador TBLPTR diante das instruções de leitura e escrita de tabelas. Somente usado com instruções de leitura de tabela (TBLRD) e escrita de tabela (TBLWT).</p> <p>Não modifica o registrador.</p> <p>Pós-incremente o registrador.</p> <p>Pós-decremente o registrador.</p> <p>Pré-incremente o registrador.</p>
Bits	
a	<p>Bit de acesso a RAM.</p> <p>Se a = 0, Access RAM (o BSR é ignorado).</p> <p>Se a = 1, RAM Bank (o BSR é usado para selecionar o banco). (padrão)</p>
b	Endereço do bit dentro de um arquivo de registro de 8bits. Ele varia de 0 a 7.
d	<p>Bit de seleção da destinação.</p> <p>Se d = 0, o resultado é armazenado no WREG.</p> <p>Se d = 1, o resultado é armazenado no registrador f. (padrão)</p>
s	<p>Bit de seleção do modo de chamada/retorno rápido.</p> <p>Se s = 0, chamada/retorno normal. (padrão)</p> <p>Se s = 1, chamada/retorno rápido. Certos registros são carregados nos shadow registers.</p>

Campo	Descrição
Nomes de registros	
BSR	<i>Bank Select Register.</i> Usado para selecionar o banco da RAM.
FSR	<i>File Select Register.</i>
PCL	<i>Byte baixo do Program Counter (PC).</i> PC<7:0>.
PCH	<i>Byte alto do Program Counter (PC).</i> PC<15:8>. Observação: Não é diretamente lido/escrito, o seu valor é atualizado através do PCLATH.
PCLATH	<i>Byte alto do Program Counter (PC).</i> PC<15:8>.
PCLATU	<i>Byte mais significativo do Program Counter (PC).</i> PC<20:16>.
PRODH	<i>Byte mais significativo do produto da multiplicação.</i>
PRODL	<i>Byte menos significativo do produto da multiplicação.</i>
STATUS	Registrador de <i>Status</i> .
TABLAT	Registro da memória de programa. 8bits.
TBLPTR	Ponteiro de 21bits para a memória de programa.
WREG	<i>Working register (acumulador).</i>
Nomes dos bits	
C, DC, Z, OV, N	<i>Bits de status da ALU (Unidade Lógica Aritmética): Carry, Digit Carry, Zero, Overflow, Negative.</i>
TO	<i>Bit de Time-out.</i>
PD	<i>Bit de Power-down.</i>
PEIE	<i>Bit de ativação da interrupção periférica.</i>
GIE, GIEL/H	<i>Bit de habilitação da interrupção global.</i>
Nome das funções do dispositivo	
MCLR	Pino externo (MCLR) de reinício do dispositivo.
PC	<i>Program Counter.</i>
TOS	Topo da Pilha (Stack).
WDT	<i>Watchdog Timer.</i>

Tabela 4.21: Instruções Assembly.

Operações de registrador orientado a byte		
Instruções		Descrição/Operação
ADDWF	f,d,a	Soma WREG com f e armazena o resultado no destino. destino = WREG + f.
ADDWFC	f,d,a	Soma WREG com f e Carry bit (C), e armazena o resultado no destino. destino = WREG + Carry bit (C) + f.
ANDWF	f,d,a	Executa um E lógico entre WREG e f, e armazena o resultado no destino. destino = WREG & f.
CLRF	f,a	Limpia f. f = 0.
COMF	f,d,a	Inverte os bits de f e armazena o resultado no destino. destino = ~f.

Operações de registrador orientado a byte		
Instruções		Descrição/Operação
CPFSEQ	f,a	Compara WREG com f e pula a próxima instrução se f = WREG. Se f = WREG PC = PC + 4 Caso contrário PC = PC + 2
CPFSGT	f,a	Compara WREG com f e pula a próxima instrução se f > WREG. Se f > WREG PC = PC + 4 Caso contrário PC = PC + 2
CPFSLT	f,a	Compara WREG com f e pula a próxima instrução se f < WREG. Se f < WREG PC = PC + 4 Caso contrário PC = PC + 2
DECf	f,d,a	Decrementa f. destino = f - 1.
DECFSZ	f,d,a	Decrementa em 1 o valor de f e pula a próxima instrução se f = 0. destino = f - 1 Se destino = 0 PC = PC + 4 Caso contrário PC = PC + 2
DCFSNZ	f,d,a	Decrementa em 1 o valor de f e pula a próxima instrução se f ≠ 0. destino = f - 1 Se destino ≠ 0 PC = PC + 4 Caso contrário PC = PC + 2
INCF	f,d,a	Incrementa f. destino = f + 1.
INCFSZ	f,d,a	Incrementa em 1 o valor de f e pula a próxima instrução se f = 0. destino = f + 1 Se destino = 0 PC = PC + 4 Caso contrário PC = PC + 2
INFSNZ	f,d,a	Incrementa em 1 o valor de f e pula a próxima instrução se f ≠ 0. destino = f + 1 Se destino ≠ 0 PC = PC + 4 Caso contrário PC = PC + 2

Operações de registrador orientado a byte		
Instruções		Descrição/Operação
IORWF	f,d,a	Executa um OU lógico entre WREG e f, e armazena o resultado no destino. destino = WREG f.
MOVF	f,d,a	Move o valor de f para o destino. destino = f.
MOVFF	f,f"	Move o valor de f para f". f" = f.
MOVWF	f,a	Move o valor de WREG para f. f = WREG.
MULWF	f,a	Multiplica o valor de WREG com f e armazena o resultado da operação nos registradores PRODH:PRODL. PRODH:PRODL = WREG * f.
NEGF	f,a	Nega o valor de f. f = - f.
RLCF	f,d,a	Rotaciona f para a esquerda e carrega o Carry bit.
RLNCF	f,d,a	Rotaciona f para a esquerda sem carregar o Carry bit.
RRCF	f,d,a	Rotaciona f para a direita e carrega o Carry bit.
RRNCF	f,d,a	Rotaciona f para a direita sem carregar o Carry bit.
SETF	f,a	Coloca todos os bits de f em 1. f = 0xFF.
SUBFWB	f,d,a	Subtrai f de WREG com empréstimo. destino = WREG - f - C.
SUBWF	f,d,a	Subtrai WREG de f. destino = f - WREG.
SUBWFB	f,d,a	Subtrai WREG de f com empréstimo. destino = f - WREG - C.
SWAPF	f,d,a	Inverte os nibles de f. destino<7:4> = f<3:0>, destino<3:0> = f<7:4>
TSTFSZ	f,a	Verifica o valor de f. Se for igual a 0, pula a próxima instrução. Se f = 0 PC = PC + 4 Caso contrário PC = PC + 2
XORWF	f,d,a	Executa um OU exclusivo entre WREG e f, e armazena o resultado no destino. destino = WREG ^ f.
Operações de registrador orientado a bit		
Instruções		Descrição/Operação
BCF	f,b,a	Limpa o bit b do registrador f. f = 0.
BSF	f,b,a	Seta o bit b do registrador f. f = 1.
BTFSC	f,b,a	Verifica o bit b do registrador f. Se for 1, pula a próxima instrução. Se f = 0 PC = PC + 4 Caso contrário PC = PC + 2

		Operações de registrador orientado a bit
Instruções		Descrição/Operação
BTFS	f,b,a	<p>Verifica o bit b do registrador f. Se for 0, pula a próxima instrução.</p> <p>Se $f = 1$</p> $PC = PC + 4$ <p>Caso contrário</p> $PC = PC + 2$
BTG	f,b,a	<p>Altera o bit b do registrador f.</p> $ = \sim f$
Operações de controle		
Instruções		Descrição/Operação
BC	n	<p>Desvia se <i>Carry bit</i> for igual a 1 (C=1).</p> <p>Se C=1</p> $PC = PC + 2 + 2*n$ <p>Caso contrário</p> $PC = PC + 2$
BN	n	<p>Desvia se <i>Negative bit</i> for igual a 1 (N=1).</p> <p>Se N=1</p> $PC = PC + 2 + 2*n$ <p>Caso contrário</p> $PC = PC + 2$
BNC	n	<p>Desvia se <i>Carry bit</i> for igual a 0 (C=0).</p> <p>Se C=0</p> $PC = PC + 2 + 2*n$ <p>Caso contrário</p> $PC = PC + 2$
BNN	n	<p>Desvia se <i>Negative bit</i> for igual a 0 (N=0).</p> <p>Se N=0</p> $PC = PC + 2 + 2*n$ <p>Caso contrário</p> $PC = PC + 2$
BNOV	n	<p>Desvia se <i>Overflow bit</i> for igual a 0 (OV=0).</p> <p>Se OV=0</p> $PC = PC + 2 + 2*n$ <p>Caso contrário</p> $PC = PC + 2$
BNZ	n	<p>Desvia se <i>Zero bit</i> for igual a 0 (Z=0).</p> <p>Se Z=0</p> $PC = PC + 2 + 2*n$ <p>Caso contrário</p> $PC = PC + 2$
BOV	n	<p>Desvia se <i>Overflow bit</i> for igual a 1 (OV=1).</p> <p>Se OV=1</p> $PC = PC + 2 + 2*n$ <p>Caso contrário</p> $PC = PC + 2$

		Operações de controle
Instruções		Descrição/Operação
BRA	n	Desvia incondicionalmente. $PC = PC + 2 + 2*n$
BZ	n	Desvia se Zero bit for igual a 1 ($Z=1$). Se $Z=1$ $PC = PC + 2 + 2*n$ Caso contrário $PC = PC + 2$
CALL	n,s	Chama a sub-rotina. $TOS = PC + 4$, $PC<20:1> = n$ Se $s=1$ $WREG = WREGs$, $STATUS = STATUSs$, $BSR = BSRs$ não são alterados.
CLRWDT		Limpa o Watchdog Timer (WDT). $WDT = 0$, $WDT\ postscaler = 0$, $\overline{TO} = 1$, $\overline{PD} = 1$
DAW		Ajuste decimal do WREG. Se $WREG<3:0> > 9$ ou DC = 1, $WREG<3:0> = WREG<3:0> + 6$ Caso contrário $WREG<3:0> = WREG<3:0>$ Se $WREG<7:4> > 9$ ou C = 1, $WREG<7:4> = WREG<7:4> + 6$ Caso contrário $WREG<7:4> = WREG<7:4>$
GOTO	n	Desvia o programa para o endereço n. $PC<20:1> = n$.
NOP		Sem operação.
POP		Retira um valor do topo da Stack (TOS). $TOS = TOS - 1$.
PUSH		Coloca o valor do PC+2 no topo da Stack (TOS). $TOS = PC + 2$.
RCALL	n	Chamada relativa. $TOS = PC + 2$, $PC = PC + 2 + 2*n$
RESET		Reinicia o dispositivo via software. É o mesmo que reiniciar o dispositivo através do MCLR.
RETFIE	s	Retorna da interrupção e habilita as interrupções. $PC = TOS$, $GIE/GIEH = 1$ ou $PEIE/GIEL$. Se $s=1$ $WREG = WREGs$, $STATUS = STATUSs$, $BSR = BSRs$, $PCLATU/PCLATH$ Não são alterados.
RETURN	s	Retorna de uma sub-rotina. $PC = TOS$. Se $s=1$ $WREG = WREGs$, $STATUS = STATUSs$, $BSR = BSRs$, $PCLATU/PCLATH$ Não são alterados.

Operações de controle	
Instruções	Descrição/Operação
SLEEP	Entra no modo <i>Sleep</i> . WDT = 0, WDT postscaler = 0, TO = 1, PD = 0
Operações com literal	
Instruções	Descrição/Operação
ADDLW	Soma kk com o WREG e armazena o resultado no WREG . WREG = WREG + kk.
ANDLW	Executa um E lógico entre WREG e kk, e armazena o resultado no WREG WREG = WREG & kk.
IORLW	Executa um OU lógico entre WREG e kk, e armazena o resultado no WREG . WREG = WREG kk.
LFSR	Move o kk para FSRr . FSRr = kk.
MOVLB	Move o kk para BSR<3:0> . BSR = k.
MOVLW	Move o kk para WREG . WREG = kk.
MULLW	Multiplica o valor de WREG com kk e armazena o resultado da operação nos registradores PRODH:PRODL. PRODH:PRODL = WREG * kk.
RETLW	Retorna com o valor kk no registrador WREG . WREG = kk.
SUBLW	Subtrai o WREG de kk e armazena o resultado no WREG WREG = kk - WREG .
XORLW	Executa um OU exclusivo lógico entre WREG e kk, e armazena o resultado no WREG . WREG = WREG ^ kk
Operações com memória	
Instruções	Descrição/Operação
TBLRD	Leitura da tabela. TABLAT = TBLPTR
TBLRD*+ POSTINC	Leitura da tabela com incremento após a leitura. TABLAT = TBLPTR → TBLPTR = TBLPTR + 1.
TBLRD*- POSTDEC	Leitura da tabela com decremento após a leitura TABLAT = TBLPTR → TBLPTR = TBLPTR - 1.
TBLRD*+ PREINC	Leitura da tabela com incremento antes da leitura. TBLPTR = TBLPTR + 1 → TABLAT = TBLPTR .
TBLWT	Escrita na tabela. TBLPTR = TABLAT .
TBLWT*+ POSTINC	Escrita na tabela com incremento após a leitura. TBLPTR = TABLAT → TBLPTR = TBLPTR + 1.
TBLWT*- POSTDEC	Escrita na tabela com decremento após a leitura. TBLPTR = TABLAT → TBLPTR = TBLPTR - 1.
TBLWT*+ PREINC	Escrita na tabela com incremento antes da leitura. TBLPTR = TBLPTR + 1 → TBLPTR = TABLAT .

Operações com instruções estendidas		
Observação: As instruções são válidas somente para o modo estendido.		
Instruções		Descrição/Operação
ADDFSR	f,k	Soma k com FSR e armazena o resultado no registrador FSR. FSR = FSR + k.
ADDULNK	k	Soma k com FSR2 e armazena o resultado no registrador FSR2, e retorna. FSR2 = FSR2 + k, PC = TOS.
CALLW		Chama a sub-rotina usando o registrador WREG (W). TOS = PC + 2, PCL = W, PCH = PCLATH, PCU = PCLATU
MOVSF	z',f"	Move z' para f". f" = FSR2 + z'.
MOVSS	z',z"	Move z' para z". FSR2 + z" = FSR2 + z'.
PUSHL	k	Armazena k no FSR2 e decremente FSR2. FSR2 = k, FSR2 = FSR2 - 1.
SUBFSR	f,k	Subtração do FSR. FSR(f) = FSR(f - k).
SUBULNK	k	Subtração do FSR2 e retorna. FSR2 = FSR2 - k, PC = TOS.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550

#pragma udata minha_ram=0x200
    unsigned char cont;//Declara uma variável do tipo unsigned char. [0x200]
#pragma udata

void main( ) //Função principal
{
    _asm
    start:
        MOVLW 10 //WREG = 10
        MOVLB 2 //BSR = 2 -> Seleciona o banco 2, pois o endereço 0x200 pertence ao banco 2.
        MOVWF cont,1 //cont = 10
    loop:
        DCFSNZ cont, 1 //Decrementa em 1 a variável cont e verifica se o valor de cont é diferente de zero.
        GOTO start //Executa se cont == 0
        BRA loop //Executa se cont != 0
    endasm
}
```

O exemplo anterior move (**MOVLW**) o valor 10 para o acumulador (**WREG**). Em seguida, seleciona o banco 2 (**MOVLB 2**), pois a variável *cont* está localizada no endereço 0x200, e como a memória de dados do PIC18 está distribuída em bancos com 256 (0x00 a 0xFF) registradores cada, implica que o endereço 0x200 está localizado no banco 2 (**BSR=2**). Esta localização é essencial quando se está programando em **Assembly**, pois é necessário especificar o local onde se encontra a variável a ser manipulada; caso contrário, podem ocorrer erros.

Dando continuidade à interpretação deste código, o próximo comando move (**MOVWF**) o conteúdo do acumulador (**WREG**) para a variável *cont* (localizado na *RAM Bank*) e depois decrementa o valor da variável *cont* e pula a próxima instrução se *cont* for diferente de zero (**DCFSNZ**). Enquanto o valor de *cont* for diferente de zero, a instrução **BRA** desvia o contador para o rótulo *loop*, senão o programa é desviado para o rótulo **start** (**GOTO start**).

4.21 Funções de Controle do Processador

4.21.1 Clrwdt

Essa função limpa o conteúdo do *Watchdog Timer* (WDT).

Exemplo

`Clrwdt();`

4.21.2 Descrição dos Resets

As funções listadas nesta seção podem ser usadas para ajudar a determinar a fonte causadora do *Reset/Wake-up* do dispositivo e reconfigurar o status do processador após o *Reset*. A utilização das funções listadas neste tópico requer a introdução da diretiva `#include <reset.h>` dentro do código de programa.



Se BOR ou WDT forem usados no programa, então é necessário inserir as instruções `#define BOR_ENABLED` e `#define WDT_ENABLED` no arquivo `reset.h`. Se o dispositivo estiver configurado para reiniciar por causa do *stack overflow/underflow*, então é necessário inserir a instrução `#define STVR_ENABLED` no arquivo `reset.h`.

4.21.2.1 isBOR ()

Retorna 1 se o dispositivo foi reiniciado por causa do *Brown-out Reset* (BOR). As condições dos *bits* de *Status* são:

- $\overline{\text{POR}} = 1$; $\overline{\text{BOR}} = 0$

4.21.2.2 isLVD ()

Retorna 1 se o dispositivo foi reiniciado por causa da baixa tensão de alimentação (*Low Voltage Detect*). Essa condição é verdadeira quando a tensão de alimentação é inferior ao valor especificado no registrador **LVDCON** (LVLD3:LVDL0).

4.21.2.3 isMCLR ()

Retorna 1 se o dispositivo foi reiniciado pelo pino **MCLR** durante a execução normal do programa. As condições dos *bits* de *Status* são:

- $\overline{\text{POR}} = 1$; $\overline{\text{PD}} = 1$
- Se o *Brown-out* estiver habilitado: $\overline{\text{BOR}} = 1$
- Se o WDT estiver habilitado: $\overline{\text{TO}} = 1$
- Se o *Stack overflow/underflow reset* estiver habilitado: **STKPTR <7:6>** será limpo (STKFUL = 0 e STKUNF = 0).

4.21.2.4 isPOR ()

Retorna 1 se for detectado que o microcontrolador saiu do *Power-on Reset*. As condições dos bits de *Status* são:

- $\overline{\text{POR}} = 0$; $\overline{\text{BOR}} = 0$; $\overline{\text{TO}} = 1$; $\overline{\text{PD}} = 1$



Esta condição ocorre quando o dispositivo é reiniciado a partir do pino $\overline{\text{MCLR}}$ e quando a instrução CLRWDT é executada. Após $\text{isPOR}()$, deve ser chamada a função $\text{StatusReset}()$ para selecionar os bits $\overline{\text{POR}}$ e $\overline{\text{BOR}}$.

4.21.2.5 isWDTTO ()

Retorna 1 se o dispositivo foi reiniciado por causa do estouro de tempo do *Watchdog Timer* (WDT). As condições dos bits de *Status* são:

- $\overline{\text{POR}} = 1$; $\overline{\text{BOR}} = 1$; $\overline{\text{TO}} = 0$; $\overline{\text{PD}} = 1$

4.21.2.6 isWDTWU ()

Retorna 1 se o dispositivo saiu do modo Sleep por meio do *Watchdog Timer* (WDT). As condições dos bits de *Status* são:

- $\overline{\text{POR}} = 1$; $\overline{\text{BOR}} = 1$; $\overline{\text{TO}} = 0$; $\overline{\text{PD}} = 0$

4.21.2.7 isWU ()

Retorna 1 se o dispositivo saiu do modo Sleep através do pino $\overline{\text{MCLR}}$ ou por uma interrupção. As condições dos bits de *Status* são:

- $\overline{\text{POR}} = 1$; $\overline{\text{BOR}} = 1$; $\overline{\text{TO}} = 1$; $\overline{\text{PD}} = 0$

4.21.3 Funções de Atraso

As funções de atraso definidas pelo compilador MPLAB® C18 geram um atraso baseado no ciclo de máquina ($T_{CY} \approx 4/F_{osc}$). A utilização das funções listadas neste tópico requer a introdução da diretiva `#include < delays.h >` dentro do código de programa, conforme o modelo apresentado pela função `Delay1TCY`.

Sintaxe

```
Delay1TCY( )
Delay10TCYx( valor_10x )
Delay100TCYx( valor_100x )
Delay1KTCYx( valor_1kx )
Delay10KTCYx( valor_10kx )
```

Sendo:

- **valor_10x**: valor de 8bits. '0' corresponde a um atraso de 2.560 Tcy e 1-255 gera um atraso de $valor_10x * 10$ Tcy.
- **valor_100x**: valor de 8bits. '0' corresponde a um atraso de 25.600 Tcy e 1-255 gera um atraso de $valor_100x * 100$ Tcy.

- **valor_1kx:** valor de 8bits. '0' corresponde a um atraso de 256.000 Tcy e 1-255 gera um atraso de $\text{valor_1kx} \times 1000 \text{Tcy}$.
- **valor_10kx:** valor de 8bits. '0' corresponde a um atraso de 2.560.000 Tcy e 1-255 gera um atraso de $\text{valor_10kx} \times 10000 \text{Tcy}$.

Observe que somente a função **Delay1TCY** não permite atribuição de valores. Ela gera um atraso de somente um ciclo de instrução (Tcy).

A relação entre ciclos de instrução (Ciclos_de_instrução) e tempo em segundos (T_{segundos}) pode ser obtida pela equação seguinte.

$$\text{Ciclos_de_instrução} = \frac{T_{\text{segundos}} * \text{Fosc}}{4}$$

Suponha que o microcontrolador esteja utilizando um cristal de 8MHz (Fosc) sem PLL. Logo, temos que o tempo de ciclo de instrução (Tcy) corresponde a 2MHz. Então, para gerar um delay de 30ms, serão necessários 60.000 ciclos de máquina, conforme mostra o seguinte cálculo.

$$\text{Ciclos_de_instrução} = \frac{0,03 * 8M}{4} = 60000$$

Pode-se concluir que a função mais adequada para gerar 30ms é **Delay10KTCYx** (6), pois o valor do parâmetro resulta em um ciclo de instrução múltiplo de 10.000.

Exemplo

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <delays.h> //Adiciona a biblioteca de funções de atraso.
#include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída.
void main (void)
{
    Delay1KTCYx(1); //Gera um atraso de 1*1000 ciclos de instrução.
    printf ("Teste Delay."); //Imprime a mensagem na janela Output.
}
```

4.21.4 Nop

Executa uma operação NOP (*No Operation*); isso implica que será gerado um atraso equivalente a um ciclo de instrução.

Exemplo

```
Nop();
```

4.21.5 Reset

O comando **Reset()** força o reinício do microcontrolador e pode ser utilizado em qualquer parte do programa. A utilização das funções listadas neste tópico requer a introdução da diretiva **#include <reset.h>** dentro do código de programa, como pode ser observado no exemplo a seguir.

Exemplo

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <reset.h> //Adiciona a biblioteca de funções de reset.
void main (void)
{
    unsigned char cont; //Declara uma variável do tipo int.
```

```

for (cont = 0 ; cont < 7 ; cont++)
{
    if (cont == 6)
    { Reset(); } //Reinicia o dispositivo.
}
}

```

4.21.6 Sleep

O comando **Sleep()** coloca o microcontrolador no modo de baixo consumo até que um evento externo específico o "acorde".

Exemplo

```
Sleep(); //Coloca o dispositivo no modo sleep.
```

4.22 Arquivos do Autor

4.22.1 Memória EEPROM Interna

A biblioteca de manipulação da memória EEPROM interna do microcontrolador utilizado neste livro está disponível no site da Editora Érica (www.editoraerica.com.br) com o nome de **biblioteca_eeprom_interna.h**. Vejamos na sequência a lista de comandos disponibilizados pelo arquivo.

4.22.1.1 escreve_mem_EEPROM ()

Escreve no endereço da memória EEPROM apontado por **endereco_reg**.

Sintaxe

```
escreve_mem_EEPROM ( endereco_reg, dados_EEPROM, quant_dado_escreve )
```

Sendo:

- **endereco_reg**: endereço da célula da memória EEPROM.
- **dados_EEPROM**: ponteiro para o vetor contendo os dados. (memória de dados)
- **quant_dado_escreve**: quantidade de *bytes* que devem ser escritos.

4.22.1.2 le_mem_EEPROM ()

Retorna o conteúdo da memória EEPROM apontado por **endereco_reg**.

Sintaxe

```
dado = le_mem_EEPROM ( endereco_reg )
```

Sendo:

- **endereco_reg**: endereço da célula da memória EEPROM.
- **dado**: valor de *8bits*.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include "C:\pic18\Biblioteca_eeprom_interna.h" //Biblioteca contendo as funções de manipulação da EEPROM interna.
```

```

/*A primeira célula da memória EEPROM interna do PIC18 está localizada no endereço 0xF00000 e pode ser
pré-configurada por meio do seguinte comando:*/
#pragma romdata overlay minha_eeprom=0xF00000 //Endereço da EEPROM - 0xF00000
    rom far char mem_eeprom[ ]="TESTE MEMORIA EEPROM";//Armazena a string na memória EEPROM
#pragma romdata

void main( ) //Função principal
{
    unsigned char digito_rec { 3 } = { "PIC" };
    unsigned char res;

    //Armazena uma string de três caracteres na memória EEPROM interna, iniciando pela posição 0x35.
    escreve_mem_EEPROM ( 0x35, digito_rec, 3 );
    res = le_mem_EEPROM ( 0x35 ); //Retorna o caractere presente na posição 0x35. res = 'P'.
}

```

4.22.2 Memória Flash Interna

A biblioteca de manipulação da memória Flash interna do microcontrolador utilizado neste livro está disponível no site da Editora Érica (www.editoraerica.com.br) com o nome `biblioteca_flash_interna.h`. Vejamos na sequência a lista de comandos disponibilizados pelo arquivo.

4.22.2.1 escreve_mem_flash ()

Escreve no endereço da memória Flash apontado por `endereco_reg`.

Sintaxe

`escreve_mem_flash (endereco_reg, dados_rom, quant_dado_escreve)`

Sendo:

- `endereco_reg`: endereço do registrador.
- `dados_rom`: ponteiro para o vetor contendo os dados.
- `quant_dado_escreve`: quantidade de *bytes* que devem ser escritos.



A escrita na memória de programa é executada em blocos de 32*bytes* (16 *words*) cada. Para averiguar os endereços livres da memória Flash, basta compilar o programa e visualizar o mapa de memória no **View → Program memory**.

4.22.2.2 le_mem_flash ()

Retorna o conteúdo da memória Flash apontado por `endereco_reg`.

Sintaxe

`dado = le_mem_flash (endereco_reg)`

Sendo:

- `endereco_reg`: endereço do registrador.
- `dado_rom`: ponteiro para o vetor contendo os dados.
- `quant_dado_le`: quantidade de *bytes* que devem ser lidos.

```

// Arquivo de cabeçalho do PIC18F4550.
#include <pic18/biblioteca_flash_interna.h> //Biblioteca contendo as funções de manipulação da Flash interna.
//Função principal:
void ler_dados[36] = {0x23,0x98,0x18,0x33,0x47,0x78,0x98,0x18,0x33,0x47,0x78,0x98,0x18,0x33,0x47,
0x78,0x98,0x18,0x33,0x47,0x78,0x98,0x18,0x33,0x47,0x78,0x98,0x18,0x33,0x47,0x60,0x60,0x60,
0x60,0x30,
void ler_cont_mem_flash[36],
void short long ponteiro_rom;
//Lê os dados presentes no buffer de dados para dentro da memória de programa, iniciando pelo endereço
//TBLPTR e retorna o valor do ponteiro TBLPTR: ponteiro_rom = 0x003040*
void rom=escreve_item_flash(0x3000,dados,36);
//Leitura dos 36 bytes presentes na memória de programa, iniciando pelo endereço 0x3000.
item_flash(0x3000 cont_mem_flash,36);

```

4.23 Dicas

A seguir orienta sobre os cuidados que devem ser tomados para evitar erros na hora da compilação, bem como garantir que o microcontrolador funcione adequadamente. Os erros podem ser minimizados; basta seguir um simples modelo sugerido.

1. Incluir bibliotecas.
 2. Incluir as diretivas relacionadas à configuração do dispositivo.
 3. Declarar variáveis e constantes globais.
 4. Incluir definições
 5. Declarar funções secundárias
 - a) Declarar variáveis e constantes locais.
 - b) Escrever o restante da rotina.
 6. Declarar uma função principal.
 - a) Declarar variáveis e constantes locais.
 - b) Configurar os periféricos do dispositivo.
 - c) Escrever o restante da rotina.

Vejamos agora um pequeno exemplo de código de programa utilizando o modelo supracitado.

```

1 → #include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.

// Fosc = 20MHz
// Tciclo = 4/Fosc = 0.2us
2 → #pragma config FOSC = HS
2 → #pragma config CPUDIV = OSC1_PLL2

2 → #pragma config WDT = OFF      //Desabilita o Watchdog Timer (WDT)
2 → #pragma config PWRT = ON     //Habilita o Power-up Timer (PWRT).
2 → #pragma config BOR = ON      //Brown-out Reset (BOR) habilitado somente no hardware.
2 → #pragma config BORV = 1       //Voltagem do BOR é 4,33V.
2 → #pragma config PBADEN = OFF   //RB0,1,2,3 e 4 configurado como I/O digital
2 → #pragma config LVP = OFF     //Desabilita o Low Voltage Program

4 → #define B_LED_1 PORTBbits RB5 //Define outro nome para a estrutura
4 → #define B_RELÉ PORTBbits RB7 //Define outro nome para a estrutura
4 → #define LED_1 PORTEbits RE0 //Define outro nome para a estrutura.
4 → #define RELÉ PORTEbits RE2 //Define outro nome para a estrutura.

void main(void)
{
6a → TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída
6b → TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída
6b → TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
6b → TRISD = 0b00000000; //RD0 a RD7 – saída
6b → TRISE = 0b00000000; //RE0 a RE2 – saída

6c → while(1) //Looping infinito
    {
        if (B_LED_1 == 0)//Verifica se o valor presente no pino RB5 é 0v (botão B_3 pressionado)
        { LED_1 = 1 } //Liga o LED_1.
        else
        { LED_1 = 0 } //Desliga o LED_1.

        if (B_RELÉ == 0)//Verifica se o valor presente no pino RB7 é 0v. (botão B_1 pressionado)
        { RELÉ = 1. } //Liga o RELÉ
        else
        { RELÉ = 0. } //Desliga o RELÉ
    }
}

```

Figura 4.5: Modelo de código de programa.

Exercícios

1. De acordo com os identificadores listados em seguida.
 - I. Var_3
 - II. 3_var
 - III. @cont
 - IV. _cont
 - V. Teste_esp
 - VI. Var-3

Assinale a alternativa correta.

- a) I, II V e VI estão corretas.
 - b) I, V e VI estão corretas.
 - c) I, IV e V estão corretas.
 - d) I, IV, V e VI estão corretas.
 - e) II, III e IV estão corretas.
2. Suponha que uma variável do tipo **unsigned char** (8bits) receba um valor igual a 600. Podemos afirmar que o valor presente na variável é igual a:
 - a) 600
 - b) 88
 - c) 344
 - d) 0
 - e) 89
- 3.

'o'	'd'	't'
'e'	'b'	'c'
'F'	'Z'	'a'

```
char M_teste[3][3]={'o','d','t','e','b','c','F','Z','a'};
char c_lido;
c_lido=M_teste[ L ][ C ]; //c_lido='c'
```

Quais os valores que os parâmetros L e C devem assumir para que o caractere da variável c_lido seja igual a 'c'?

- a) L=1 e C=3
- b) L=2 e C=3
- c) L=2 e C=1
- d) L=3 e C=2
- e) L=1 e C=2



Microcontrolador PIC18F4550

Este capítulo faz uma breve introdução ao microcontrolador PIC18F4550, o qual é largamente utilizado nos exemplos e projetos propostos no livro, abordando os principais conceitos relacionados ao seu modo de funcionamento.

Deste ponto em diante, diversos circuitos eletrônicos são sugeridos para o desenvolvimento dos projetos. É importante salientar que os circuitos apresentados têm aplicação didática e precisam ser aperfeiçoados, caso deseje utilizá-los para fins comerciais.

5.1 Introdução

O PIC18F4550 é construído com base na arquitetura *Harvard* com instruções do tipo RISC (Computador com Conjunto Reduzido de Instruções). É um dispositivo de 8bits dotado de 32Kbytes de memória de programa e 2.048bytes de memória RAM. Esse dispositivo pode ser alimentado com tensões entre 4V e 5.5V, além de operar em frequência de até 48MHz (12 MIPS - milhões de instruções por segundo). Ele pode ser alimentado diretamente por um oscilador de 48MHz ou por um cristal associado com o bloco PLL. Além disso, possui um oscilador interno de 8MHz, que pode ser derivado em 8MHz, 4MHz, 2MHz, 1MHz, 500KHz, 250KKHz, 125KHz e 31KHz.

Esse modelo possui 40 pinos, dos quais 35 podem ser configurados como I/O, e diversos periféricos, tais como memória EEPROM de 256bytes, um módulo CCP e ECCP, um módulo SPI e I²C, treze conversores A/D de 10bits de resolução com tempo de aquisição programável, dois comparadores analógicos, uma comunicação EUSART, um TIMER de 8bits (TIMER2) e três de 16bits (TIMER0, TIMER1 e TIMER3), um módulo de detecção de alta/baixa voltagem (HLVD), além de ter um módulo USB 2.0 capaz de operar no modo *low-speed* (1.5Mbps) ou *full-speed* (12Mbps).

5.1.1 Memórias

Existem basicamente três tipos de memória nos microcontroladores PIC18, sendo memória de programa, memória de dados e memória EEPROM.

A memória de programa (do tipo **não-volátil**) está presente em todos os dispositivos. É responsável pelo armazenamento das instruções (código de programa) e pode ser adquirida no formato FLASH (F), ROM (CR) ou OTP/EPROM (C).

A memória de dado (do tipo **volátil**) também está presente em todos os dispositivos e retém dados do *Special Function Registers* (SFRs) e *General Purpose Registers* (GPRs). Os SFRs são relacionados à configuração, controle e status dos periféricos e portas E/S, enquanto os GPRs armazenam e manipulam os dados do usuário.

O terceiro tipo de memória é a EEPROM (do tipo não-volátil). Ela permite armazenar e manipular dados, porém o conteúdo não é perdido, caso o dispositivo não esteja sendo alimentado.

5.1.2 Ciclo de Máquina

O microcontrolador PIC18 também conta com a tecnologia PIPELINE, que resulta no aumento da velocidade de processamento, pois ao mesmo tempo que uma instrução é executada, a próxima instrução é localizada e carregada no registro de instrução (IR) em um ciclo de máquina. Devido a esse comportamento é possível afirmar que uma instrução é efetivamente executada em um único ciclo de máquina.

O ciclo de máquina do microcontrolador PIC18 é dividido em quatro fases: Q1, Q2, Q3 e Q4, resultando em uma velocidade de ciclo de máquina equivalente a um quarto do valor do oscilador.

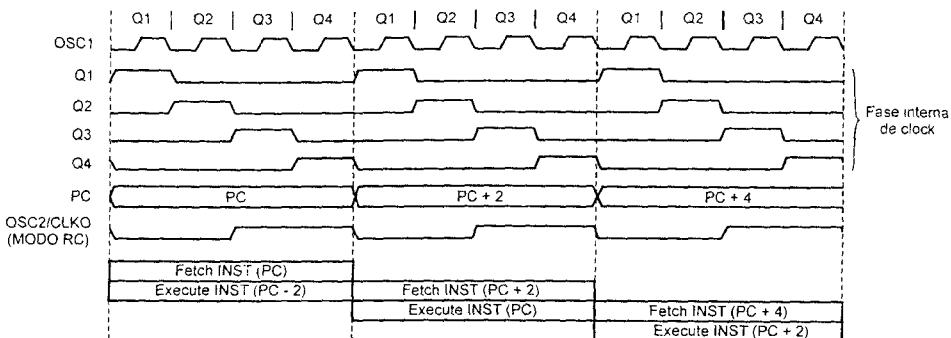


Figura 5.1: Ciclo de máquina.

Observando a Figura 5.1, veja que cada ciclo de máquina executa dois eventos paralelamente, os quais vamos chamar de ciclo de localização e ciclo de execução.

O ciclo de localização (*Fetch*), internamente, incrementa o *Program Counter* (PC) durante o ciclo Q1, e a instrução localizada na memória de programa é carregada para dentro do registro de instrução (IR) durante o ciclo Q4.

No ciclo de execução (*Execute*) a instrução localizada é carregada para dentro do registro de instrução (IR) durante o ciclo Q1 e é decodificada e executada durante os ciclos Q2 a Q4. A memória de dado é lida em Q2 e escrita em Q4.

Exemplo

Se o *clock* do microcontrolador for de 20MHz (Fosc=20MHz), o ciclo de máquina será equivalente a:

$$f_{\text{maq}} = \frac{F_{\text{oosc}}}{4} = \frac{20\text{MHz}}{4} = 5\text{MHz}$$

Esta frequência corresponde a um ciclo de máquina igual a:

$$T_{\text{maq}} = \frac{1}{f_{\text{maq}}} = \frac{1}{5\text{MHz}} = 0.2\mu\text{s}$$

5.2 Pinagem

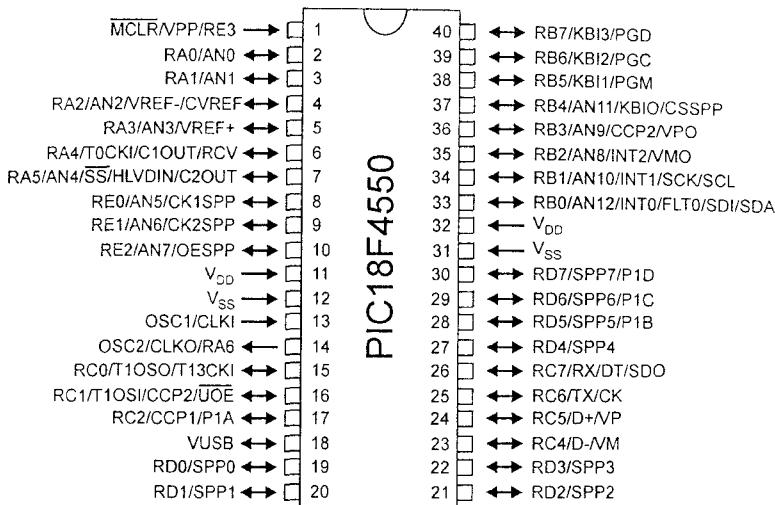


Figura 5.2: Pinagem do PIC18F4550.

Tabela 5.1: Nomenclatura dos pinos.

Pino	Tipo	Nome	Descrição
1	ST — ST	MCLR V _{PP} RE3	Master Clear Reset. Entrada de voltagem de programação (12V). Entrada Digital.
2	TTL Analógico	RA0 AN0	I/O Digital. Canal analógico 0.
3	TTL Analógico	RA1 AN1	I/O Digital. Canal analógico 1.
4	TTL Analógico Analógico Analógico	RA2 AN2 Vref- CVref	I/O Digital. Canal analógico 2. Entrada de tensão negativa de referência (Baixa) para o conversor A/D. Saída de Vref do comparador analógico.
5	TTL Analógico Analógico	RA3 AN3 Vref+	I/O Digital. Canal analógico 3. Entrada de tensão positiva de referência (Alta) para o conversor A/D.
6	ST ST — TTL	RA4 T0CKI C1OUT RCV	I/O Digital. Entrada de clock externo para o Timer0. Saída do comparador 1. Entrada RCV para o transceptor externo USB.

Pino	Tipo	Nome	Descrição
7	TTL Analógico	RA5 AN4	I/O Digital. Canal analógico 4.
	TTL Analógico	SS	Entrada de seleção, no modo SPI escravo.
	—	HLVDIN	Entrada do módulo <i>High/Low-Voltage Detect</i> (HLVD).
	—	C2OUT	Saída do comparador 2.
8	ST Analógico	RE0 AN5	I/O Digital. Canal analógico 5.
	—	CK1SPP	Saída de clock 1 do SPP.
9	ST Analógico	RE1 AN6	I/O Digital. Canal analógico 6.
	—	CK2SPP	Saída de clock 2 do SPP.
10	ST Analógico	RE2 AN7	I/O Digital. Canal analógico 7.
	—	OESPP	Saída de habilitação do SPP.
11	—	V _{DD}	Alimentação positiva.
12	—	V _{ss}	Terra (GND).
13	Analógico Analógico	OSC1 CLKI	Cristal oscilador Entrada de clock externo.
14	— — TTL	OSC2 CLKO RA6	Cristal oscilador Saída de clock. I/O Digital.
15	ST — ST	RC0 T1OSO T13CKI	I/O Digital. Saída do oscilador do TIMER1. Entrada de clock externo para o TIMER1 / TIMER3.
16	ST CMOS ST —	RC1 T1OSI CCP2* UOE	I/O Digital. Entrada do oscilador do TIMER1. Entrada do Capture2, Saída do Compare2, Saída do PWM2. Saída OE para o transceptor externo USB.
17	ST ST TTL	RC2 CCP1 P1A	I/O Digital. Entrada do Capture1, Saída do Compare1, Saída do PWM1. Saída A do PWM do módulo ECCP1.
18	—	VUSB	Saída de tensão do regulador interno (USB 3.3V).
19	ST TTL	RD0 SPP0	I/O Digital. I/O de dado 0. (SPP).
20	ST TTL	RD1 SPP1	I/O Digital. I/O de dado 1. (SPP).
21	ST TTL	RD2 SPP2	I/O Digital. I/O de dado 2. (SPP).
22	ST TTL	RD3 SPP3	I/O Digital. I/O de dado 3. (SPP).
23	TTL — TTL	RC4 D- VM	I/O Digital. Linha D- do barramento USB. Entrada VM para o transceptor externo USB.

Pino	Tipo	Nome	Descrição
24	TTL	RC5	I/O Digital.
	—	D+	Linha D- do barramento USB.
	TTL	VP	Entrada VP para o transceptor externo USB.
25	ST	RC6	I/O Digital.
	—	TX	Transmissão assíncrona EUSART.
	ST	CK	Clock síncrono EUSART1.
26	ST	RC7	I/O Digital.
	ST	RX	Recepção assíncrona EUSART.
	ST	DT	Dados síncronos EUSART.
	—	SDO	Saída de dado do módulo SPI.
27	ST	RD4	I/O Digital.
	TTL	SPP4	I/O de dado 4. (SPP).
28	ST	RD5	I/O Digital.
	TTL	SPP5	I/O de dado 5. (SPP).
	—	P1B	Saída B do PWM do módulo ECCP1.
29	ST	RD6	I/O Digital.
	TTL	SPP6	I/O de dado 6. (SPP).
	—	P1C	Saída C do PWM do módulo ECCP1.
30	ST	RD7	I/O Digital.
	TTL	SPP7	I/O de dado 7. (SPP).
	—	P1D	Saída D do PWM do módulo ECCP1.
31	—	V _{SS}	Terra (GND).
32	—	V _{DD}	Alimentação positiva
33	TTL	RB0	I/O Digital.
	ST	INT0	Interrupção externa 0.
	ST	FLT0	Entrada de falha do Enhanced PWM.
	Analógico	AN12	Canal analógico 12.
	ST	SDI	Entrada de dados do módulo SPI.
34	ST	SDA	I/O de dados do módulo I ² C.
	TTL	RB1	I/O Digital.
	ST	INT1	Interrupção externa 1.
	Analógico	AN10	Canal analógico 10.
	ST	SCK	I/O de clock serial síncrono do módulo SPI.
35	ST	SCL	I/O de clock serial síncrono do módulo I2C.
	TTL	RB2	I/O Digital.
	ST	INT2	Interrupção externa 2.
36	Analógico	AN8	Canal analógico 8.
	—	VMO	Saída VMO para o transceptor externo USB.
36	TTL	RB3	I/O Digital.
	Analógico	AN9	Canal analógico 9.
	ST	CCP2**	Entrada do Capture2, Saída do Compare2, Saída do PWM2.
	—	VPO	Saída VPO para o transceptor externo USB.

Pino	Tipo	Nome	Descrição
37	TTL	RB4	I/O Digital.
	Analógico	AN11	Canal analógico 11.
	TTL	KBI0	Pino de interrupção na mudança de estado.
	—	CSSPP	Saída de seleção do chip. (SPP)
38	TTL	RB5	I/O Digital.
	TTL	KBI1	Pino de interrupção na mudança de estado.
	ST	PGM	Ativa a programação ICSP em baixa voltagem.
39	TTL	RB6	I/O Digital.
	TTL	KBI2	Pino de interrupção na mudança de estado.
	ST	PGC	In-circuit debugger e <i>clock</i> de programação ICSP.
40	TTL	RB7	I/O Digital.
	TTL	KBI3	Pino de interrupção na mudança de estado.
	ST	PGD	In-circuit debugger e dados de programação ICSP.

Legenda: * = Modulo CCP2 padrão. Bit CCP2MX (#pragma config CCP2MX = ON).

** = Módulo CCP2 alternativo. Bit CCP2MX (#pragma config CCP2MX = OFF)

ST = Entrada Schmitt Trigger compatível com nível CMOS.

TTL = Entrada compatível com nível TTL.

CMOS = Entrada/Saída compatível com nível CMOS.

Analógico = Entrada analógica.



As portas B e D do microcontrolador PIC18F4550 possuem *pull-up* internos, que podem ser habilitados através dos bits RBPU (INTCON2<7> = 0) e RDPU (PORTE<7> = 1), respectivamente, desde que os pinos estejam configurados como entrada.

5.3 Diagrama de Blocos do PIC18F4550

Como este capítulo apresenta as principais funções do microcontrolador PIC18F4550, não podemos deixar de comentar a estrutura interna desse poderoso modelo de microcontrolador.

O diagrama de blocos do PIC18F4550 está representado na Figura 5.3.

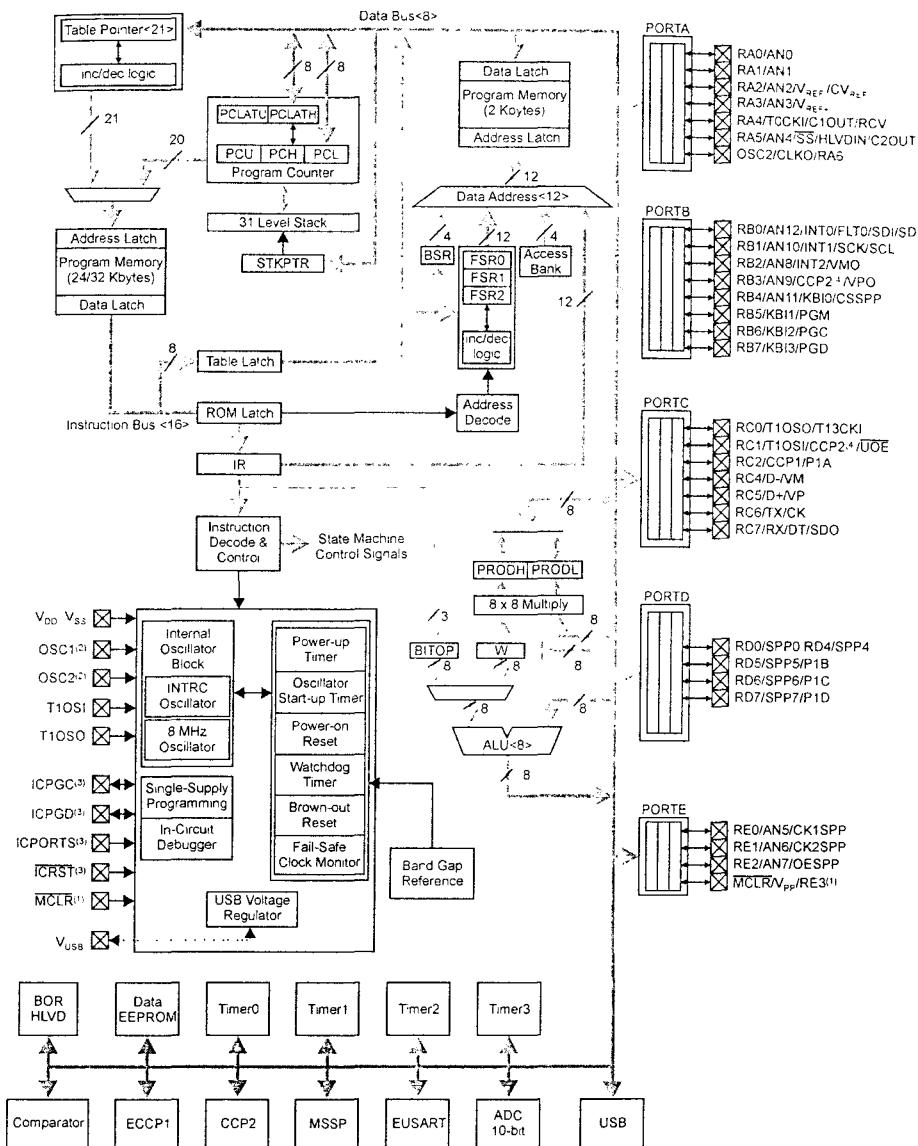


Figura 5.3: Diagrama de blocos do PIC18F4550.

Logo ao centro, temos a unidade de processamento (ALU em inglês, ou ULA em português) responsável pela execução das operações lógicas e aritméticas, além do 8 x 8 hardware multiplier que faz parte da ULA, e é destinado a executar operações de multiplicação, cujo resultado é armazenado nos registros PRODH e PRODL. O registro relacionado à ULA é o Work Register (WREG), também conhecido como acumulador.

À esquerda estão os circuitos internos, tais como bloco de oscilador interno, bloco de programação/debugação, regulador de tensão da USB, *Power-up Timer* (PWRT), *Oscillator Start-up Timer* (OST), *Power-on Reset* (POR), *Watchdog Timer* (WDT), *Brown-out Reset* (BOR) e *Fail-Safe Clock Monitor* (FSCM).

Na parte superior, é possível visualizar o bloco da memória de dados de 2Kbytes (RAM), cujos endereços dos dados podem ser acessados com o auxílio dos registros *Bank Select Register* (**BSR**), *Access Bank* (**a**) ou *File Select Registers* (**FSR0**, **FSR1**, **FSR2**).

No canto superior esquerdo temos a memória de programa de 32Kbytes do tipo FLASH, endereçada por um ponteiro de 21bits (*Program Counter* - PC) representado pelos registros **PCU:PCH:PCL**, sendo que **PCU** e **PCH** não são acessados diretamente pelo usuário e sim por intermédio do **PCLATU** e **PCLATH** respectivamente. Logo abaixo do *Program Counter* temos a pilha (Stack) de 31 níveis controlada pelo registro **STKPTR**.

Do lado direito encontram-se as portas de I/O (PORTA, PORTB, PORTC, PORTD e PORTE) e na parte inferior do diagrama estão os periféricos, tais como *High/Low-Voltage Detect* (HLVD), memória de dados EEPROM, TIMERS (TIMER0, 1, 2 e 3), comparador analógico, módulo CCP1 e CCP2, módulo MSSP, EUSART, conversor A/D de 10bits e USB.

Não se preocupe com os termos utilizados, pois todos serão comentados no decorrer do livro.

5.4 Memória de Dados

A memória de dados do microcontrolador PIC18, também chamada de arquivo de registro, está implementada como RAM estática. Ela compreende os conteúdos dos *Special Function Registers* (SFR) e *General Purpose Registers* (GPR).

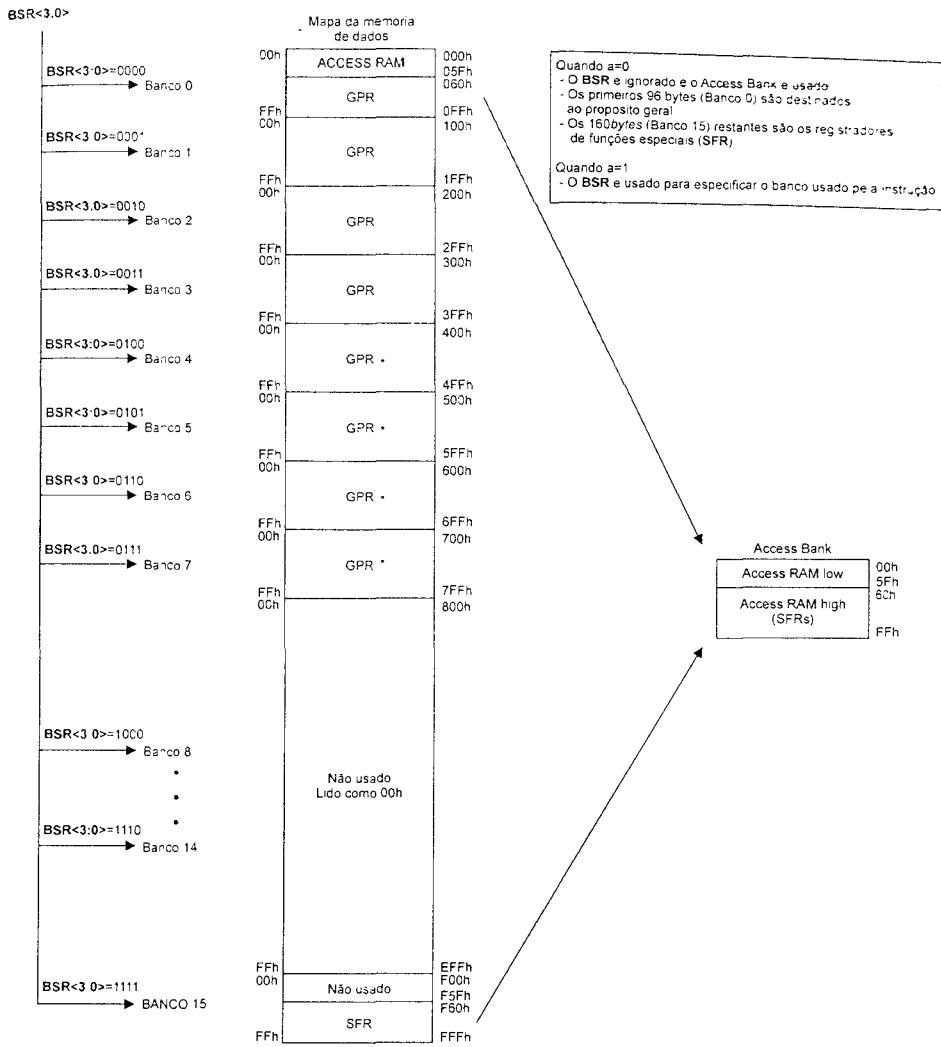
Os SFRs são destinados ao armazenamento de informações concernentes ao controle e status do controlador, além das funções relacionadas aos periféricos, enquanto os GPRs são comumente utilizados para reter conteúdos dos dados utilizados pelo usuário.

O arquivo de registro está distribuído em 16 bancos, e cada um contém 256bytes. Para acessar o registro de modo direto, é necessário selecionar o banco através do *Bank Select Register* (**BSR<3:0>** - 4bits menos significativos) e em seguida enviar o endereço do registro (8bits). O acesso indireto da memória de dados não requer o uso do **BSR**, e pode ser feito por meio dos *File Select Registers* (**FSR0**, **FSR1** e **FSR2**), os quais podem ser utilizados como ponteiros para os locais da memória, em que são efetuadas instruções de escrita/leitura. Os FSRs são formados pelos pares **FSRnH** (4bits menos significativos) e **FSRnL** (8bits), totalizando 12bits.

A memória de dados do PIC18 também conta com uma seção especial denominada de *Access RAM* (mapeada no *Access Bank*), localizada no banco 0 (destinada ao GPR), e o banco 15 (compreende os SFRs), cujos registros podem ser acessados diretamente (8bits) sem a necessidade do **BSR**, resultando em um tempo de acesso equivalente a um ciclo. O *Access RAM* do banco 0 é ideal em aplicações em que os dados necessitam ser acessados rapidamente, além de gerar códigos mais eficientes.

A região denominada de *Access RAM* está mapeada no *Access Bank*, o qual é habilitado através do *Access RAM bit* (**a**). Quando **a** é igual a 0, a instrução é forçada a usar o mapa de endereço do *Access RAM* e ignora o **BSR**. Se **a** for igual a 1, a instrução usa a seleção do banco (**BSR**) e os 8bits de endereço.

Veja em seguida o mapa de memória de dados do microcontrolador PIC18F4550.



Legenda * - Os bancos também são usados como buffer para a USB

Figura 5.4: Mapa de memória de dados do microcontrolador PIC18F4550.

5.4.1 Registradores de Funções Especiais (SFRs)

	Endereço do arquivo		Endereço do arquivo		Endereço do arquivo		Endereço do arquivo	
TOSU	FFFh	INDF2*	FDFh	CCPR1H	FBFh	IPR1	F9Fh	UEP15
TOSH	FFEh	POSTINC2*	FDEh	CCPR1L	FBEh	PIR1	F9Eh	UEP14
TOSL	FFDh	POSTDEC2*	FDDh	CCP1CON	FBDh	PIE1	F9Dh	UEP13
STKPTR	FFCh	PREINC2*	FDCh	CCPR2H	FBCh	-	F9Ch	UEP12
PCLATU	FFBh	PLUSW2*	FDBh	CCPR2L	FBBh	OSCTUNE	F9Bh	UEP11
PCLATH	FFAh	FSR2H	FDAh	CCP2CON	FBAh	-	F9Ah	UEP10
PCL	FF9h	FSR2L	FD9h	-	FB9h	-	F99h	UEP9
TBLPTRU	FF8h	STATUS	FD8h	BAUDCON	FB8h	-	F98h	UEP8
TBLPTRH	FF7h	TMR0H	FD7h	ECCP1DEL	FB7h	-	F97h	UEP7
TBLPTRL	FF6h	TMR0L	FD6h	ECCP1AS	FB6h	TRISE	F96h	UEP6
TABLAT	FF5h	TOCON	FD5h	CVRCON	FB5h	TRISD	F95h	UEP5
PRODH	FF4h	-	FD4h	CMCON	FB4h	TRISC	F94h	UEP4
PROOL	FF3h	OSCCON	FD3h	TMR3H	FB3h	TRISB	F93h	UEP3
INTCON	FF2h	HLVDCON	FD2h	TMR3L	FB2h	TRISA	F92h	UEP2
INTCON2	FF1h	WDTCON	FD1h	T3CON	FB1h	-	F91h	UEP1
INTCON3	FF0h	RCON	F00h	SPBRGH	FB0h	-	F90h	UEP0
INFO*	FEFh	TMR1H	FCFh	SPBRG	FAFh	-	F8Fh	UCFG
POSTINC0*	FEEh	TMR1L	FCEh	RCREG	FAEh	-	F8Eh	UADDR
POSTDEC0*	FEDh	T1CON	FCDh	TXREG	FADh	LATE	F8Dh	UCON
PREINCO*	FECh	TMR2	FCCh	TXSTA	FACh	LATD	F8Ch	USTAT
PLUSW0*	FEBh	PR2	FCBh	RCSTA	FABh	LATC	F8Bh	UEIE
FSR0H	FEAh	T2CON	FCAh	-	FAAh	LATB	F8Ah	UEIR
FSR0L	FE9h	SSPBUF	FCh	EEADR	FA9h	LATA	F89h	UIE
WREG	FE8h	SSPADD	FC8h	EEDATA	FA8h	-	F88h	UIR
INDF1*	FE7h	SSPSTAT	FC7h	EECON2*	FA7h	-	F87h	UFRMH
POSTINC1*	FE6h	SSPCON1	FC6h	EECON1	FA6h	-	F86h	UFRML
POSTDEC1*	FE5h	SSPCON2	FC5h	-	FA5h	-	F85h	SPPCON
PREINC1*	FE4h	ADRESH	FC4h	-	FA4h	PORTE	F84h	SPPEPS
PLUSW1*	FE3h	ADRESL	FC3h	-	FA3h	PORTD	F83h	SPPCFG
FSR1H	FE2h	ADCON0	FC2h	IPR2	FA2h	PORTC	F82h	SPPDATA
FSR1L	FE1h	ADCON1	FC1h	PIR2	FA1h	PORTB	F81h	-
BSR	FE0h	ADCON2	FC0h	PIE2	FA0h	PORTA	F80h	-

□ Locais não implementados

* Não é um registrador físico.

Figura 5.5: Mapa do Registrador de Funções Especiais (SFR).

Como já mencionado, os SFRs armazenam informações relacionadas ao controle e status do controlador e as funções relacionadas ao gerenciamento dos periféricos. Esses registradores estão presentes no banco 15 da memória de programa do PIC18F4550, e são apresentados na sequência.

As colunas POR e BOR das tabelas apresentadas a partir deste ponto informam os valores dos registradores quando o dispositivo sofre um Power-on Reset (POR) ou Brown-out Reset (BOR).

Banco 15										
Endereço	Nome	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	POR E BOR
FFFh	TOSU	-	-	-	Byte mais significativo da pilha (Stack) (TOS<20 15>)					---0 0000
FFEh	TOSH				Byte alto da pilha (Stack) (TOS<15 8>)					0000 000C
FFDh	TOSL				Byte baixo da pilha (Stack) (TOS<7 0>)					0000 0000
FFCh	STKPTR	STKFUL	STKUNF	-		Ponteiro de retorno da pilha				00 0 0000
FFBh	PCLATU	-	-	-	Byte mais significativo do PC<20 15>					---0 0000
FFAh	PCLATH				Byte alto do PC<15 8>					0000 0000
FF9h	PCL				Byte baixo do PC<7 0>					0020 0000
FF8h	TBLPTRU	-	-	bit 21*	Byte mais significativo do ponteiro da tabela de Memória de Programa (TBLPTR<20 16>)					-00 0000
FF7h	TBLPTRH				Byte alto do ponteiro da tabela de Memória de Programa (TBLPTR<15 8>)					0000 0000
FF6h	TBLPTRL				Byte baixo do ponteiro da tabela de Memória de Programa (TBLPTR<7 0>)					0000 0000
FF5h	TABLAT				Registro da Memória de Programa					0000 0000
FF4h	PRODH				Byte mais significativo do resultado de uma operação de multiplicação					xxxx xxxx
FF3h	PRODL				Byte menos significativo do resultado de uma operação de multiplicação					xxxx xxxx
FF2h	INTCON	GIE/GIEH	FEIE/GIEL	TMRCIE	INTOIE	RBIE	TMROIF	INTOIF	RBIF	0000 000x
FF1h	INTCON2	RBU	INTEDG0	INTEDG1	INTEDG2	-	TMROIP	-	RBIP	1111 -1-1
FF0h	INTCON3	INT2IP	INT1IP	-	INT2IE	INT1IE	-	INT2IF	INT1IF	11-0 0-00
FEFh	INDFO				Memória de dados endereçada indiretamente pelo FSR0 **					-----
FEEh	POSTINCO				Memória de dados endereçada indiretamente pelo FSR0, o qual é incrementado após a operação **					-----
FEDh	POSTDEC0				Memória de dados endereçada indiretamente pelo FSR0, o qual é decrementado após a operação **					-----
FECh	PREINC0				Memória de dados endereçada indiretamente pelo FSR0, o qual é pré-incrementado **					-----
FE8h	PLUSW0				Memória de dados endereçada indiretamente pelo FSR0 com valor de offset do registrador W **					-----
FEAh	FSR0H	-	-	-	Byte mais significativo do ponteiro 0 do endereço da memória de dados (acesso indireto)					---- 0000
FE9h	FSROL				Byte menos significativo do ponteiro 0 do endereço da memória de dados (acesso indireto)					xxxx xxxx
FE8h	WREG				Registrador W					xxxx xxxx
FE7h	INDF1				Memória de dados endereçada indiretamente pelo FSR1 **					-----
FE6h	POSTINC1				Memória de dados endereçada indiretamente pelo FSR1, o qual é incrementado após a operação **					-----
FE5h	POSTDEC1				Memória de dados endereçada indiretamente pelo FSR1, o qual é decrementado após a operação **					-----
FE4h	PREINC1				Memória de dados endereçada indiretamente pelo FSR1, o qual é pré-incrementado **					-----
FE3h	PLUSW1				Memória de dados endereçada indiretamente pelo FSR1 com valor de offset do registrador W **					-----
FE2h	FSR1H	-	-	-	Byte mais significativo do ponteiro 1 do endereço da memória de dados (acesso indireto)					---- 0000
FE1h	FSR1L				Byte menos significativo do ponteiro 1 do endereço da memória de dados (acesso indireto)					xxxx xxxx
FE0h	BSR	-	-	-	Ponteiro para o banco (bank)					---- 0000
FDFh	INDF2				Memória de dados endereçada indiretamente pelo FSR2 **					-----
FDEh	POSTINC2				Memória de dados endereçada indiretamente pelo FSR2, o qual é incrementado após a operação **					-----
FDDh	POSTDEC2				Memória de dados endereçada indiretamente pelo FSR2, o qual é decrementado após a operação **					-----
FDCh	PREINC2				Memória de dados endereçada indiretamente pelo FSR2 o qual é pré-incrementado **					-----
FDBh	PLUSW2				Memória de dados endereçada indiretamente pelo FSR2 com valor de offset do registrador W **					-----
FDAh	FSR2H	-	-	-	Byte mais significativo do ponteiro 2 do endereço da memória de dados (acesso indireto)					---- 0000

Legenda x = desconhecido * = bit 21 do registrador TBLPTRU permite o acesso aos bits de configuração do dispositivo,
** - Não é um endereço físico, - = não está implementado

Figura 5.6: Banco 15.

Banco 15										
Endereço	Nome	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	POR E BOR
FD9h	FSR2L					Byte menos significativo do ponteiro 2 do endereço da memória de dados (acesso indireto)				xxxx xxxx
FD8h	STATUS	-	-	-	N	OV	Z	DC	C	--x xxxx
FD7h	TMR0H					Byte mais significativo do TIMER0				0000 0000
FD6h	TMR0L					Byte menos significativo do TIMER0				xxxx xxxx
FD5h	T0CON	TMR0ON	T0BBIT	TOCS	TOSE	PSA	T0PS2	T0PS1	T0PS0	1111 1111
FD4h	-									-----
FD3h	OSCCON	IDLEN	IRCF2	IRCF1	IRCF0	OSTS	IOFS	SCS1	SCS0	0100 0000
FD2h	HVLVDCON	VDIRMAG	-	IRVST	HVLVDEN	HVLVDL3	HVLVDL2	HVLVDL1	HVLVDL0	0-00 0101
FD1h	WDTCON	-	-	*	-	-	-	-	SWDTEN	---- --O
FD0h	RCON	IPEN	SBOREN	-	RI	TO	PO	POR	BOR	0q-1 11q0
FCFh	TMR1H					Byte mais significativo do TIMER1				xxxx xxxx
FC Eh	TMR1L					Byte menos significativo do TIMER1				xxxx xxxx
FCDn	T1CON	RD15	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON	0000 0000
FCCh	TMR2					Registrador do TIMER2				0000 0000
FCBn	PR2					Registrador do período do TIMER2				1111 1111
FDAh	T2CON	-	T2OUTPS3	T2OUTPS2	T2OUTPS1	T2OUTPS0	TMR2ON	T2CKPS1	T2CKPS0	000 0000
FC5h	SSPBUF					Buffer de transmissão/recepção do módulo SSP				xxxx xxxx
FC6h	SSPADD					Registrador do endereço no modo I2C Slave ou registrador do baud rate no modo Master				0000 0000
FC7h	SSPSSTAT	SMP	CKE	D/A	P	S	R/W	UA	BF	0000 0000
FC8h	SSPCON1	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	0000 0000
FC9h	SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN	0000 0000
FC4h	ADRESH					Byte mais significativo do valor retornado pelo conversor A/D				xxxx xxxx
FC3h	ADRESL					Byte menos significativo do valor retornado pelo conversor A/D				xxxx xxxx
FC2h	ADCON0	-	-	CHS3	CHS2	CHS1	CHS0	GO/DONE	ADON	--0 0000
FC1h	ADCON1	-	-	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0	--0 0qqq
FCCh	ADCON2	ADFM	-	ACQT2	ACQT1	ACQ0	ADCS2	ADCS1	ADCS0	0-00 0000
FBFh	CCPR1H					Byte mais significativo do registrador do módulo CCP1				xxxx xxxx
FBEh	CCPR1L					Byte menos significativo do registrador do módulo CCP1				xxxx xxxx
FB0h	CCP1CON	P1M1	P1M0	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0	0000 0000
FBCh	CCPR2H					Byte mais significativo do registrador do módulo CCP2				xxxx xxxx
FBBh	CCPR2L					Byte menos significativo do registrador do módulo CCP2				xxxx xxxx
FBAh	CCP2CON	-	-	DC2B1	DC2B0	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--0 0000
FB9h	-									-----
FB8h	BAUDCON	ABDOVF	RCIDL	-	SCKP	BRG16	-	WUE	ABDEN	01-0 00
FB7h	ECCP1DEL	PRSEN	PDC6	PDC5	PDC4	PDC3	PDC2	PDC1	PDC0	0000 0000
FB6h	ECCP1AS	ECCPASE	ECCPAS2	ECCPAS1	ECCPAS0	PSSAC1	PSSAC0	PSSBD1	PSSBD0	0000 0000
FB5h	CVRCON	CVREN	CVROE	CVRR	CVRSS	CVR3	CVR2	CVR1	CVR0	0000 0000
FB4h	CMCON	C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0	0000 0111
FB3h	TMR3H					Byte mais significativo do TIMER3				xxxx xxxx
FB2h	TMR3L					Byte menos significativo do TIMER3				xxxx xxxx
FB1h	T3CON	RD15	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	T3SYNC	TMR3CS	TMR3ON	0000 0000
FB0h	SPBRGH					Byte mais significativo do gerador de baud rate da USART				0000 0000
FAFh	SPBRG					Byte menos significativo do gerador de baud rate da USART				0000 0000
FAEh	RCREG					Buffer de recepção da USART				0000 0000
FADh	TXREG					Buffer de transmissão da USART				0000 0000
FACh	TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D	0000 0010
FABh	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x
FAAh	-									-----

Legenda: ** = não é um endereço físico q = valor depende da condição, x = desconhecido - = não está implementado

Figura 5.7: Banco 15.

Banco 15										
Endereço	Nome	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	POR E BOR
FA9h	EEDADR	Registrador do endereço da EEPROM interna					xxxx xxxx			
FA8h	EEDATA	Registrador do dado da EEPROM interna					0000 0000			
FA7h	EECON2	Registrador 2 do controle da EEPROM interna **					0000 0000			
FA6h	EECON1	EEPGD	CFGSD	-	FREE	WRERR	WRREN	WR	RD	xx-0 xx00
FA5h	-	-					-----			
FA4h	-	-					-----			
FA3h	-	-					-----			
FA2h	IPR2	OSCFIP	CMIP	USBIP	EEIP	BCLIP	HLVDIP	TMR3IP	CCP2IP	1111 1111
FA1h	PIR2	OSCFIF	CMIF	USBIF	EEIF	BCLIF	HLVDIF	TMR3IF	CCP2IF	0000 0000
FA0h	PIE2	OSCFIE	CMIE	USBIE	EEIE	BCLIE	HLVDIE	TMR3IE	CCP2IE	0000 0000
F9Fh	IPR1	SPPIP	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	1111 1111
F9Eh	PIR1	SPPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000
F9Dh	PIE1	SPPIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000
F3Ch	-	-					-----			
F9Bh	OSCTUNE	INTSRC	-	-	TUN4	TUN3	TUN2	TUN1	TUN0	0-0 0000
F9Ah	-	-					-----			
F99h	-	-					-----			
F98h	-	-					-----			
F97h	-	-					-----			
F96h	TRISE	-	-	-	-	-	TRISE2	TRISE1	TRISE0	---- -111
F95h	TRISD	TRISD7	TRISD6	TRISD5	TRISD4	TRISD3	TRISD2	TRISD1	TRISD0	1111 1111
F94h	TRISC	TRISC7	TRISC6	-	-	-	TRISC2	TRISC1	TRISCO	11-- -111
F93h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111
F92h	TRISA	-	TRISA6 ⁽¹⁾	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	-1111 1111
F91h	-	-					-----			
F90h	-	-					-----			
F8Fh	-	-					-----			
F8Eh	-	-					-----			
F8Dh	LATE	-	-	-	-	-	LATE2	LATE1	LATE0	---- -xxx
F8Ch	LATD	LATD7	LATD6	LATD5	LATD4	LATD3	LATD2	LATD1	LATD0	xxxx xxxx
F8Bh	LATC	LATC7	LATC6	-	-	-	LATC2	LATC1	LATC0	xx-- -xxx
F8Ah	LATB	LATB7	LATB6	LATB5	LATB4	LATB3	LATB2	LATB1	LATB0	xxxx xxxx
F89h	LATA	-	LATA6 ⁽¹⁾	LATA5	LATA4	LATA3	LATA2	LATA1	LATA0	-xxx xxxx
F88h	-	-					-----			
F87h	-	-					-----			
F86h	-	-					-----			
F85h	-	-					-----			
F84h	PORTE	RDPUI	-	-	-	RE3 ⁽²⁾	RE2	RE1	RE0	0--- x000
F83h	PORTD	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	xxxx xxxx
F82h	PORTC	RC7	RC6	RC5 ⁽³⁾	RC4 ⁽³⁾	-	RC2	RC1	RC0	xxxx -xxx
F81h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx
F80h	PORTA	-	RA6 ⁽¹⁾	RA5	RA4	RA3	RA2	RA1	RA0	-x0 0000
F7Fh	UEP15	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F7Eh	UEP14	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F7Dh	UEP13	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F7Ch	UEP12	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F7Bh	UEP11	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000

Legenda ** = não é um endereço físico, q = valor depende da condição, x = desconhecido - = não está implementado

(1) = O pino RA6 é configurado através dos vários modos de operação do oscilador primário

(2) = O pino RE3 está disponível se o bit do MCLRE for '0'

(3) = Os pinos RC5 e RC4 estão disponíveis somente se a USB estiver desabilitada (UCON<3> = 0)

Figura 5.8: Banco 15.

Banco 15										
Endereço	Nome	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	POR E BOR
F7Ah	UEP10	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F79h	UEP9	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F78h	UEP8	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F77h	UEP7	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F76h	UEP6	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F75h	UEP5	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F74h	UEP4	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F73h	UEP3	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F72h	UEP2	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F71h	UEP1	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F70h	UEP0	-	-	-	EPHSHK	EPCONDIS	EPOUTEN	EPININ	EPSTALL	--0 0000
F6Fh	UCFG	UTEYE	UOEMON	-	UPUEN	UTRDIS	FSEN	PPB1	PPB0	00-0 0000
F6Eh	UADDR	-	ADDR6	ADDR5	ADDR4	ADDR3	ADDR2	ADDR1	ADDR0	-000 0000
F6Dn	UCON	-	PPBRST	SEO	PKTDIS	USBEN	RESUME	SUSPNO	-	-0x0 000-
F6Ch	USTAT	-	ENDP3	ENDP2	ENDP1	ENDP0	DIR	PPBI	-	-xxx xxx-
F6Bh	UEIE	BTSEE	-	-	BTOEE	DFN8EE	CRC16EE	CRC5EE	PIDEF	0-0 0000
F6Ah	UEIR	BTSEF	-	-	BTOEF	DFN8EF	CRC16EF	CRC5EF	PIDEF	0-0 0000
F69h	UIE	-	SOFIE	STALLIE	IDLEIE	TRNIE	ACTVIE	UERRIE	URSTIE	-000 0000
F68h	UIR	-	SOFIF	STALLIF	IDLEIF	TRNIF	ACTVIF	UERRIF	URSTIF	-000 0000
F67h	UFRMH	-	-	-	-	FRM10	FRM9	FRM8	-	-xxx
F66h	UFRML	FRM7	FRM6	FRM5	FRM4	FRM3	FRM2	FRM1	FRM0	xxxx xxxx
F65h	SPPCON	-	-	-	-	-	-	SPPOWN	SPPEN	-00 -00
F64h	SPPEPS	RDSPP	WRSPP	-	SSPBUSY	ADDR3	ADDR2	ADDR1	ADDR0	00-0 0000
F63h	SPPCFG	CLKCFG1	CLKCFG0	CSEN	CLK1EN	WS3 ⁽²⁾	WS2	WS1	WS0	0000 0000
F62h	SPPDATA	DATA7	DATA6	DATA5	DATA4	DATA3	DATA2	DATA1	DATA0	0000 0000
F61h	-	-	-	-	-	-	-	-	-	-
F60h	-	-	-	-	-	-	-	-	-	-

Legenda ** = não é um endereço físico, q = valor depende da condição, x = desconhecido - = não está implementado

Figura 5.9: Banco 15.

5.5 Memória de Programa e a Stack

O PIC18F4550 é constituído de uma memória de programa do tipo FLASH de 32Kbytes (0000h a 7FFFh), com espaço de 16bits (*Word Instructions*), e possui uma *Stack* (pilha) de 31 níveis, cujos endereços da memória de programa são apontados pelo *Program Counter* (PC).

O *Program Counter* é um ponteiro de 21bits que especifica o endereço da instrução que será executada e é representado por três registradores de 8bits. Os 5bits mais significativos estão localizados no registrador **PCLATU** (PC<20:16>), seguidos de 8bits intermediários **PCLATH** (PC<15:8>) e por último, os 8bits menos significativos **PCL** (PC<7:0>).

5.5.1 Memória de Programa

A memória de programa armazena o código do programa do microcontrolador, além de permitir escrever, ler e apagar os dados durante a operação normal.

O espaço da memória de dados (8bits) é diferente da memória de programa (16bits), por isso as instruções são executadas com operações de tabela. Um bloco de tabela contém um dado representado por 8bits, em vez de uma instrução (16bits), cujo endereço de um dado presente na memória de programa é indicado pelo ponteiro de tabela **TBLPTR**, distribuído em três registros: **TBLPTRU:TBLPTRH:TBLPTRL**, que juntos formam um ponteiro de 22bits. O 22ºbit permite acessar o *device ID*, o *user ID* e o *configuration bits*.

Para averiguar os endereços livres da memória FLASH, basta compilar o programa e visualizar o mapa de memória no **View → Program memory**.

Suponha que um determinado programa tenha ocupado somente até o endereço 0x04EE da memória de programa do PIC18. Sendo assim, teoricamente é possível armazenar dados a partir do endereço 0x04EF. Porém, ao introduzir um comando de escrita ou leitura de dados, o código de programa é aumentado, por este motivo dê preferência aos últimos endereços da memória de programa.

Line	Date	Address	OpCode	Label	Comment
622	14.7	04EF	0000		DATA 0x4EF
623	04E	04E	0000	RET	RET
624	04E	04E	0000	_zero_16	LEER 0, 0
625	04F	04F	0000		0000
626	04F	04F	0000		0000
627	04F	04F	0000		0000
628	04F	04F	0000		0000
629	04F	04F	0000		0000
630	04F	04F	0000		0000
631	04F	04F	0000		0000
632	04F	04F	0000		0000
633	04F	04F	0000		0000
634	04F	04F	0000		0000
635	04F	04F	0000		0000
636	04F	04F	0000		0000
637	04F	04F	0000		0000
638	04F	04F	0000		0000
639	04F	04F	0000		0000
640	04F	04F	0000		0000
641	04F	04F	0000		0000
642	04F	04F	0000		0000

Figura 5.10: Memória de programa.

5.5.2 Valores

O microcontrolador PIC18 também conta com um vetor de Reset (0000h) e um de interrupção (0008h a 0018h). O vetor de Reset é apontado pelo PC sempre que o dispositivo for iniciado ou reiniciado. Como essa família de microcontrolador dispõe de várias fontes de interrupção, ele permite definir se elas serão tratadas como alta prioridade (0008h) ou baixa prioridade (0018h) no atendimento, ou seja, é possível definir o nível de prioridade no atendimento às interrupções, setando o bit IPEN (RCON<7> = 1).

5.5.3 Stack (Pilha)

A pilha (Stack) não faz parte da memória de programa nem da memória de dados. É um local que armazena o endereço de retorno quando instruções de desvio são executadas, ou uma interrupção é reconhecida. Todo desvio do programa faz com que a pilha seja carregada com o valor do *Program Counter* (PC). Deste modo, quando uma instrução de retorno é executada, o PC recebe o endereço do topo da pilha e a decremente; logo, o programa continua a ser executado exatamente a partir do ponto em que a operação foi interrompida.

A Stack do microcontrolador PIC18 permite a combinação de 31 chamadas do programa e ocorrência de interrupção, e o ponteiro associado a ela é de 5bits (**STKPTR<4:0>**) e sempre iniciado em '00000'. No entanto, somente o topo da pilha é acessível e o endereço de retorno está dividido em três registradores dispostos na sequência **TOSU:TOSH:TOSL**.

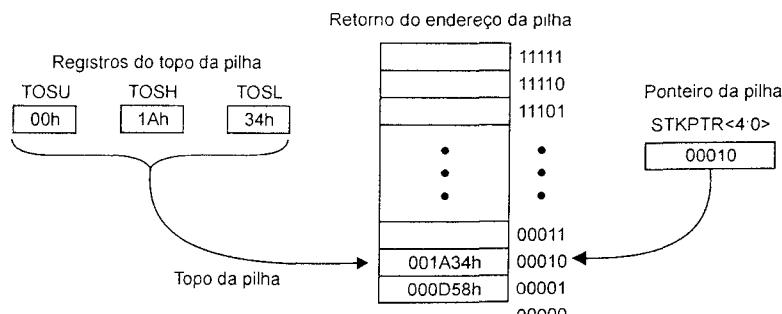


Figura 5.11: Pilha do microcontrolador PIC18.

A pilha (*Stack*) é incrementada sempre que uma instrução de desvio (**CALL** ou **RCALL**) é executada. Essa instrução incrementa o ponteiro da *Stack* (**STKPTR<4:0>**) e coloca o valor do PC dentro da pilha, enquanto a instrução de retorno (**RETURN**, **RETLW** ou **RETFIE**) transfere o conteúdo indicado pelo ponteiro da *Stack* para o PC, e depois a decrementa.

Se o ponteiro da *Stack* estiver na posição 31 e ocorrer uma chamada (**CALL**), o bit **STKFUL** (**STKPTR<7>**) é selecionado, indicando que a pilha está cheia, e se ela for disparada uma quantidade de vezes suficiente para descarregá-la por completo, o bit **STKUNF** (**STKPTR<6>**) é selecionado. Sempre que um desses dois eventos ocorre, é possível configurar o dispositivo para entrar no estado de *Reset*, selecionando o bit **STVREN** (**#pragma config STVREN = ON**); caso contrário, o ponteiro da *Stack* continua na posição 31 e o endereço contido na posição apontada não é alterado.



Os bits **STKFUL** e **STKUNF** devem ser limpos por software ou por um POR.

5.5.4 Verificação e Proteção do Código do Programa

A memória de programa do microcontrolador PIC18 está dividida em cinco blocos, sendo *Boot block*, *Block 0,1,2 e 3*. Cada bloco possui três bits associados à proteção do código, cujas funções são:

- Bit de proteção do código. Impede a leitura/escrita externa (CPn)
- Bit de proteção contra escrita (WRTn)
- Bit de proteção contra a instrução de leitura executada fora do bloco (EBTRn)

A Figura 5.12 mostra os blocos da memória de programa do PIC18F4550.

Faixa de endereço							
000000h	0007FFh	000800h	001FFFh	002000h	003FFFh	004000h	005FFFh
Boot Block	Block 0	Block 1	Block 2	Block 3	Não implementado	Lido como '0'	1FFFFFFh

Figura 5.12: Blocos da memória de programa do PIC18F4550.



A alteração dos bits CPn, WRTn e EBTRn é explicada no Capítulo 6.

5.6 Oscilador

Os microcontroladores necessitam de uma fonte de *clock* para que possam processar as informações, sendo a velocidade de processamento proporcional à frequência fornecida pela fonte de *clock*.

A CPU do microcontrolador PIC18F4550 suporta três diferentes fontes de *clock*, cuja seleção é feita através dos bits **SCS1:SCS0** (**OSCCON<1:0>**), os quais permitem as seguintes combinações:

- **OSCCON<1:0> = 1x**: oscilador interno.
- **OSCCON<1:0> = 01**: oscilador TIMER 1.
- **OSCCON<1:0> = 00**: oscilador primário.

Veja a seguir o diagrama de *clock* do microcontrolador PIC18F4550.

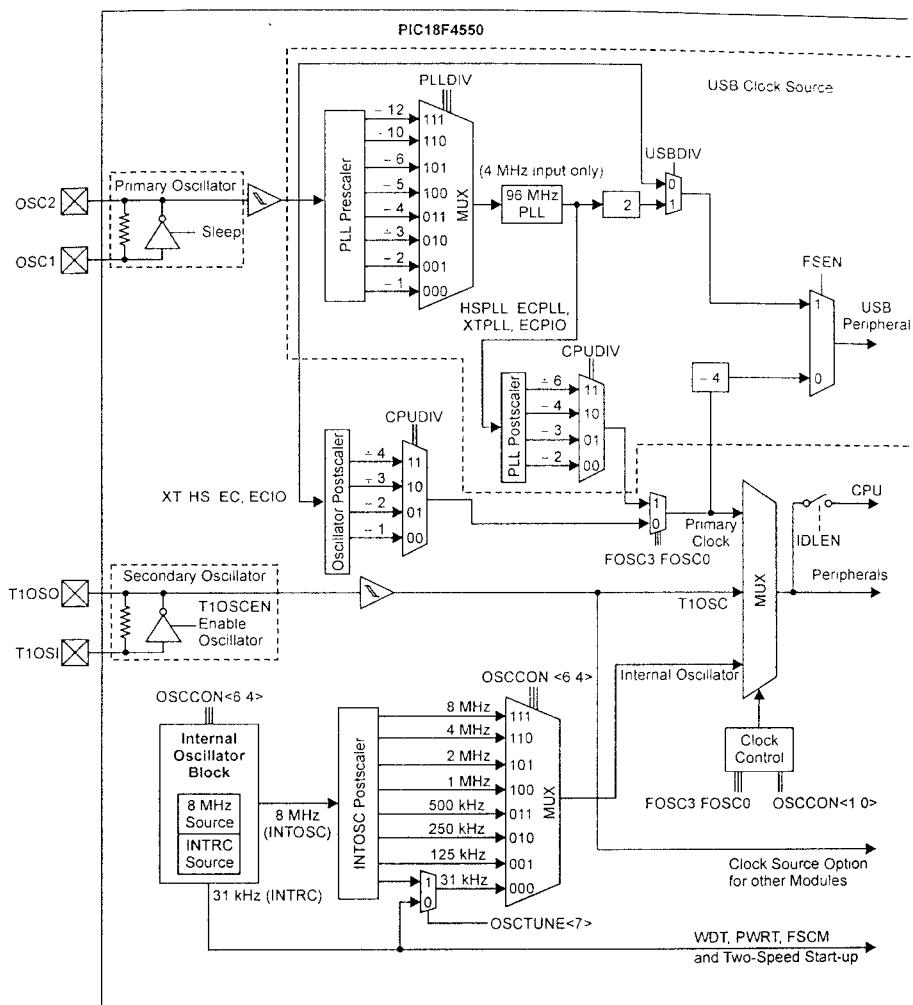


Figura 5.13: Diagrama de clock do microcontrolador PIC18F4550.

5.6.1 Oscilador Interno

Existem duas fontes de clock associadas ao bloco do oscilador interno. A principal saída desse bloco é um clock de 8MHz (INTOSC) que pode ser usado diretamente para fornecer sinal de clock para a CPU, ou então pode ser derivado em outras frequências compreendidas entre 31,25KHz e 4MHz. A frequência de 31,25Khz é selecionada pelo bit INTSRC (OSCTUNE<7> = 1).

Outra fonte de clock é o oscilador RC interno (INTRC), o qual fornece uma frequência de 31Khz e é selecionada através do bit INTSRC (OSCTUNE<7> = 0).

Ambas as fontes de *clock* são selecionadas pelos bits IRCF2:IRCF0 (**OSCCON<6:4>**), cuja combinação fornece as seguintes frequências:

Tabela 5.2: Configuração do oscilador interno

OSCCON<6:4>	Descrição
111	8MHz (INTOSC)
110	4MHz (INTOSC)
101	2MHz (INTOSC)
100	1MHz (INTOSC)
011	500KHz (INTOSC)
010	250KHz (INTOSC)
001	125KHz (INTOSC)
000	31KHz (INTOSC ou INTRC)

O bloco do oscilador interno é calibrado pelo fabricante para fornecer frequência de 8MHz (INTOSC), no entanto ela varia com a tensão de alimentação (V_{cc}) e temperatura. Por este motivo é possível fazer um ajuste fino através dos bits TUN4:TUN0 (**OSCTUNE<4:0>**), cujos valores podem resultar nos seguintes eventos:

- **OSCTUNE<4:0> = 01111:** frequência máxima.
- ...
...
- **OSCTUNE<4:0> = 00000:** frequência ajustada pelo fabricante.
- ...
...
- **OSCTUNE<4:0> = 10000:** frequência mínima.

A estabilidade da frequência de saída do INTOSC é sinalizada pelo bit IOFS (**OSCCON<2>**), sendo 1 - Frequência estável e 0 - Frequência instável.

5.6.1.1 Modos do Oscilador Interno

Quando o módulo USB está habilitado e o *clock* do dispositivo está configurado para receber o sinal de *clock* do bloco do oscilador interno, é possível configurar o *clock* para o módulo USB de quatro maneiras.

- **#pragma config INTOSC_HS:** oscilador interno, HS usado pelo USB.
- **#pragma config INTOSC_XT:** oscilador interno, XT usado pelo USB.
- **#pragma config INTOSC_EC:** oscilador interno, CLKO (Fosc/4) sobre o pino RA6, EC usado pelo USB.
- **#pragma config INTOSCI_O_EC:** oscilador interno, pino RA6 configurado como I/O, EC usado pelo USB.

5.6.2 Oscilador Secundário

O TIMER 1 pode operar em dois níveis de consumo, sendo modo de baixo consumo (*low-power*) ou em um nível de consumo maior que o *low-power*. Quando o bit LPT1OSC é setado (**#pragma config LPT1OSC = ON**), o oscilador do TIMER 1 opera no modo de *low-power* (LP); caso contrário (**#pragma config LPT1OSC = OFF**), opera no modo em que o nível de consumo é maior (modo-padrão).

O circuito oscilador do tipo *low-power* associado ao TIMER 1 pode ser habilitado pelo bit T1OSCEN (**T1CON<3> = 1**). Esse circuito é projetado para cristais de aproximadamente 32KHz e continua funcionando independente do modo de gerenciamento de energia.

Se o usuário desejar usar o oscilador do TIMER 1 para prover sinal de *clock* para o sistema, o sistema é normalmente iniciado com a configuração de oscilador interno (INTRC), e dentro do código, os registros T1CON e OSCCON devem ser configurados de modo que o *clock* para o sistema passe a ser fornecido pela fonte de *clock* TIMER 1.

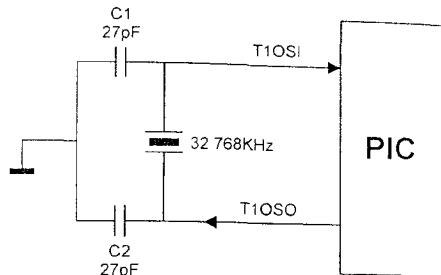


Figura 5.14: Circuito oscilador LP do TIMER1.

5.6.3 Oscilador Primário

O microcontrolador PIC® suporta diversos tipos de osciladores externos, e dentre os mais conhecidos estão HS, XT, intRC e EC.

- **HS (High Speed):** dispõe do maior nível de condução, ele é designado para cristais e ressonadores cuja frequência é igual ou superior a 4MHz. Não podemos esquecer que o consumo de energia é diretamente proporcional à frequência de trabalho. Isso significa que quanto maior for a frequência de trabalho maior será o consumo de energia.
- **XT (Xtal):** é destinado a cristais e ressonadores de 1MHz a 4MHz. Esse modo está relacionado ao consumo moderado de energia, uma vez que consome menos energia que no modo **HS**. Os ressonadores normalmente se enquadram no modo **HS**, pois costumam consumir mais corrente que os cristais. Se o ressonador utilizado estiver apresentando problemas no modo **XT**, modifique o modo do oscilador para **HS**.

Vimos que os cristais e ressonadores são capazes de fornecer sinais precisos de *clock*, por isso são largamente utilizados em microcontroladores. No entanto, para que o cristal funcione adequadamente, é necessário introduzir um par de capacitores (C1 e C2).

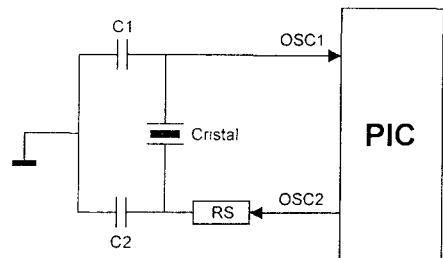


Figura 5.15: Cristal ou ressonador.

O par de capacitores sugeridos pela Microchip para o modelo PIC18F4550 está listado nas tabelas a seguir. Os valores sugeridos para os capacitores não são os ideais para qualquer tipo de cristal, e sim para os cristais utilizados na realização dos experimentos, porém, servem como um ponto de partida.

Tabela 5.3: Cristal x Capacitor.

Tipo de oscilador	Frequência do cristal	Capacitores (C1 e C2)
XT	4MHz	27 pF
	4MHz	27 pF
	8MHz	22 pF
	20MHz	15 pF

Tabela 5.4. Ressonador x Capacitor.

de oscilador	Frequência do cristal	Capacitores (C1 e C2)
XT	4MHz	33 pF
HS	8MHz	27pF
	16MHz	22pF

Para verificar se o resistor RS é necessário ou não, basta verificar se o sinal do pino OSC2 apresenta um formato senoidal. Se o sinal estiver cortado, ou seja, tenha alcançado a máxima amplitude, então deve-se inserir o resistor RS para reduzir a condução de corrente e, consequentemente, baixar a amplitude do sinal, garantindo um sinal senoidal perfeito.

- **intRC (Internal Resistor and Capacitor):** possui a mesma funcionalidade do modo RC padrão. A principal diferença é que o circuito RC está implementado dentro do dispositivo, ou seja, o microcontrolador dispõe de um oscilador RC interno (não são todos os modelos que dispõem desse circuito). Esse modo apresenta o menor custo, já que não há necessidade de um circuito externo para fornecer sinal de clock para o microcontrolador. Sendo assim, os pinos que eram destinados à entrada de sinal de clock podem ser utilizados para desempenhar outras funções.
- **EC (External Oscillator):** é selecionado quando um oscilador externo é utilizado para fornecer sinal de clock para o microcontrolador. O grande inconveniente do oscilador externo é o elevado consumo de energia. Esse modo não está disponível em todos os dispositivos, neste caso, deve-se selecionar o modo XT ou HS.

5.6.3.1 Configuração do Oscilador Externo

- **PLL desabilitado:** o oscilador primário (XT, HS, EC, ECIO) passa pelo bloco chamado *postscaler*, que permite ao usuário fornecer uma outra frequência ao dispositivo, diferente da frequência de entrada (cristal, ressonador ou oscilador externo). Esse bloco é controlado pela diretiva `#pragma config CPUDIV` que fornece as seguintes saídas:
 - **CPUDIV = OSC1_PLL2:** sinal de clock para o sistema é o oscilador primário.
 - **CPUDIV = OSC2_PLL3:** sinal de clock para o sistema é o oscilador primário dividido por 2.
 - **CPUDIV = OSC3_PLL4:** sinal de clock para o sistema é o oscilador primário dividido por 3.
 - **CPUDIV = OSC4_PLL6:** sinal de clock para o sistema é o oscilador primário dividido por 4.
- **PLL habilitado:** quando o PLL está habilitado, o oscilador primário (XTPLL, HSPLL, ECPLL, ECPIO) entra em um bloco chamado *PLL Prescaler*, controlado pelos bits PLLDIV2:PLLDIV0 (`#pragma config PLLDIV`), cuja combinação fornece uma frequência de saída igual a 4MHz para o próximo bloco (96MHz PLL). Veja em seguida a descrição desses bits.
 - **PLLDIV = 1:** oscilador primário de 4MHz.
 - **PLLDIV = 2:** oscilador primário de 8MHz dividido por 2.
 - **PLLDIV = 3:** oscilador primário de 12MHz dividido por 3.
 - **PLLDIV = 4:** oscilador primário de 16MHz dividido por 4.
 - **PLLDIV = 5:** oscilador primário de 20MHz dividido por 5.
 - **PLLDIV = 6:** oscilador primário de 24MHz dividido por 6.
 - **PLLDIV = 10:** oscilador primário de 40MHz dividido por 10.
 - **PLLDIV = 12:** oscilador primário de 48MHz dividido por 12.
 O bloco 96MHz PLL fornece uma frequência de saída equivalente a 96MHz, que entra em outro bloco chamado *PLL Postscaler*, controlado pela diretiva `#pragma config CPUDIV` que fornece as seguintes saídas:
 - **CPUDIV = OSC1_PLL2:** sinal de clock para o sistema é 96MHz dividido por 2. (Fosc = 48MHz)

- **CPUDIV = OSC2_PLL3:** sinal de *clock* para o sistema é 96MHz dividido por 3. ($\text{Fosc} = 32\text{MHz}$)
- **CPUDIV = OSC3_PLL4:** sinal de *clock* para o sistema é 96MHz dividido por 4. ($\text{Fosc} = 24\text{MHz}$)
- **CPUDIV = OSC4_PLL6:** sinal de *clock* para o sistema é 96MHz dividido por 6. ($\text{Fosc} = 16\text{MHz}$)

5.6.4 Funções do Oscilador para a USB

O módulo USB interno do PIC18F4550 pode operar no modo *full-speed* (48MHz) ou *low-speed* (6MHz). Porém, é necessário satisfazer as frequências impostas pelo módulo, a fim de garantir o seu correto funcionamento.

O modo *full-speed* é selecionado pelo bit FSEN (**UCFG<2>** = 1) e necessita de um *clock* de 48MHz que pode ser obtido através do oscilador primário de 48MHz ou pelo bloco de 96MHz PLL dividido por 2, que resulta em uma frequência de 48MHz. Essas opções de *clock* são selecionadas a partir da diretiva **#pragma config**.

- » **#pragma config USBDIV = 1:** *clock* proveniente do oscilador primário de 48MHz.
- » **#pragma config USBDIV = 2:** *clock* proveniente do bloco com saída de 96MHz dividido por 2.

O modo *low-speed* é selecionado pelo bit FSEN (**UCFG<2>** = 0) e necessita de um *clock* de 6MHz, proveniente do canal do *clock* primário ou do bloco 96MHz PLL. Por este motivo, o *clock* primário deve ser igual a 24MHz, o qual será dividido por 4 e resultará 6MHz. Veja o exemplo a seguir.

Exemplo

```
// Fosc = 48MHz e Fusb = 48MHz
#pragma config PLLDIV = 5           // Fcrystal = 20MHz -- 20MHz/5 = 4MHz.
#pragma config CPUDIV = OSC1_PLL2   // Bloco de 96MHz PLL dividido por 2 = 48MHz
#pragma config USBDIV = 2           // Clock proveniente do bloco com saída de 96MHz dividido por 2.
#pragma config FOSC = HSPLL_HS     // Oscilador HS com PLL habilitado, HS usado pelo USB.
```

5.7 Gerenciamento de Energia

O gerenciamento da energia consumida pelo microcontrolador PIC® é indispensável quando o circuito é alimentado por meio de bateria, pois circuitos dessa natureza devem consumir o mínimo de energia a fim de maximizar o tempo de carga da bateria.

O PIC18F4550 pode funcionar em três estados, conhecidos como RUN, IDLE e SLEEP.

- » **RUN:** é o modo padrão de execução, ou seja, a CPU e os periféricos estão ligados.
- » **IDLE:** desliga a CPU, porém os periféricos continuam funcionando. O microcontrolador pode sair desse estado através do *Watchdog Timer* (WDT) ou por meio de uma interrupção. Ele pode ser habilitado setando o bit IDLEN (**OSCCON<7>** = 1), assim quando a instrução **SLEEP** for chamada, o microcontrolador entrará no modo IDLE, que consome tipicamente 5,8 μA .
- » **SLEEP:** a CPU e os periféricos são desligados, desta forma o dispositivo passa a consumir uma corrente relativamente baixa, maximizando o tempo de carga da bateria. Uma sugestão para reduzir o consumo do dispositivo é colocar todos os pinos que não estão sendo utilizados como entrada e conectá-los ao V_{ss} ou V_{cc} . Outra opção é configurá-los como saída, mas não conectá-los. O dispositivo pode sair do modo SLEEP através do *Watchdog Timer* (WDT) ou por meio de uma interrupção. Ele pode ser habilitado limpando o bit IDLEN (**OSCCON<7>** = 0), assim quando a instrução **SLEEP** for chamada, o microcontrolador entrará no modo Sleep, que consome tipicamente 0,1 μA .

5.8 Reset

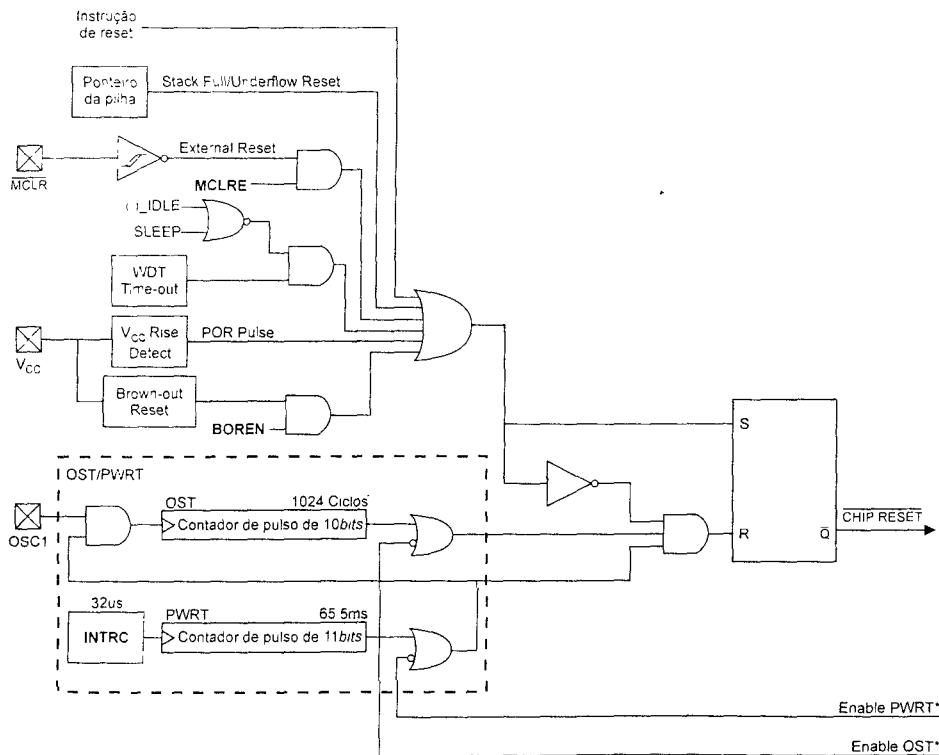
Este tópico apresenta as fontes causadoras de reinício do dispositivo e os contadores de reinício. Para descobrir a fonte que ocasionou o reset do microcontrolador, basta usar as funções listadas no tópico "DESCRÍÇÃO DOS RESETS" localizadas no Capítulo 4.

5.8.1 Fonte de Reset

Existem oito eventos especiais relacionados ao *reset* (reinício) do microcontrolador PIC18F4550, os quais podem ser classificados em internos ou externos. São eles:

1. *Power-on Reset* (POR).
2. *Brown-out Reset* (BOR) programável.
3. Reset durante a operação normal do dispositivo, através do pino MCLR.
4. Reset durante o modo de gerenciamento de energia, através do pino MCLR.
5. *Watchdog Timer* (WDT) Reset durante a operação normal do dispositivo.
6. Instrução de *Reset*.
7. *Stack full Reset*.
8. *Stack underflow Reset*.

Os eventos mencionados anteriormente são comentados no decorrer deste tópico. Veja a seguir o diagrama de blocos do circuito de *Reset* do PIC18F4550.



Legenda * - Veja a situação do time-out na Tabela 5.6

Figura 5.16: Diagrama de blocos do circuito de *Reset*.

5.8.1.1 Eventos Internos

Os eventos internos capazes de gerar o reset do dispositivo são: *Power-on Reset*, *Watchdog Timer*, *Brown-out Reset*, *Software reset instruction* e *Stack full or underflow reset*.

- **Power-on Reset (POR)**: o dispositivo é mantido em *Reset* se for detectado um aumento na tensão V_{cc} partindo de 0V, e o tempo de subida for maior que 0,05mV/ms. Ele é mantido em *Reset* até que a tensão de alimentação (V_{cc}) atinja o nível de tensão de operação do dispositivo. Sua principal função é garantir que o dispositivo esteja devidamente energizado antes de iniciar a operação.

Se a velocidade de subida for menor que 0,05mV/ms, o pino MCLR deve ser mantido em 0V por um circuito externo até que a tensão de alimentação alcance o nível de tensão necessário para a operação do dispositivo.

O bit de status POR (RCON<1>) deve ser setado via *software* após a sua ocorrência.

- **Watch Dog Timer (WDT)**: reinicia o dispositivo, caso o processador ultrapasse o limite de tempo preestabelecido pelo programador, para a execução de uma determinada rotina. O bit de sinalização é o T0 (RCON<3>).
- **Brown-out Reset (BOR)**: o dispositivo é reiniciado, se a tensão de alimentação V_{cc} for inferior ao valor de tensão predefinido pelo programador, através do bit de configuração. A faixa de tensão varia de 2,05V a 4,59V de acordo com a combinação dos bits BORV1:BORV0 (#pragma config BORV). O bit de status BOR (RCON<0>) deve ser setado via *software* após a sua ocorrência. Veja na sequência as possíveis combinações.

Tabela 5.5. Voltagem do Brown-out Reset (BOR)

<u>#pragma config BORV</u>	Tensão típica (V)
<u>#pragma config BORV = 3</u>	2.05
<u>#pragma config BORV = 2</u>	2.79
<u>#pragma config BORV = 1</u>	4.33
<u>#pragma config BORV = 0</u>	4.59

- **Software reset instruction**: o dispositivo é reiniciado pelo comando executado na linha de código. O bit de sinalização RI (RCON<4>) deve ser setado via *software* após a sua ocorrência.
- **Stack full or underflow reset**: o dispositivo é reiniciado se a pilha (stack) estiver cheia ou ocorrer um subtransbordamento da pilha, ou seja, o dispositivo será reiniciado se ocorrer um estouro da pilha (Stack full reset), ou se ela for disparada uma quantidade de vezes suficiente para descarregá-la por completo (Stack underflow reset). O bit de sinalização do Stack full reset é STKFUL (STKPTR<7>) e Stack underflow reset STKUNF (STKPTR<6>). Ambos os bits podem ser limpos por *software* ou por um POR.

5.8.1.2 Evento Externo

Há somente um evento externo que pode ocasionar o *reset* do microcontrolador PIC®. O pino relacionado a esse evento é denominado MCLR (Master Clear Reset). O MCLR é um pino de entrada do microcontrolador PIC®, que força o reinício do dispositivo sempre que for submetido a uma tensão usualmente inferior a 1V, quando o dispositivo é alimentado com 5V_{cc}. Com ele é possível realizar um *reset* externo durante o funcionamento normal do programa ou no modo *sleep*.

A Microchip recomenda o circuito apresentado na Figura 5.17 com a finalidade de evitar *resets* inesperados. Esse circuito filtra ruídos e protege o pino contra descargas eletrostáticas (ESD).

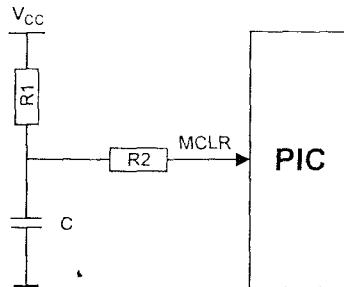


Figura 5.17: Circuito MCLR.

O resistor **R1** deve ser menor que $40\text{K}\Omega$ para garantir que a queda de tensão não exceda as especificações do dispositivo e **R2** maior que $1\text{k}\Omega$ para limitar a corrente vinda do capacitor **C** em direção ao pino MCLR.

5.3.2 Contadores de Reset do Dispositivo

O microcontrolador PIC18F4550 incorpora três contadores que ajudam a regular o processo de *Power-on Reset* do dispositivo. Esses contadores têm a finalidade de garantir que o sinal de *clock* vindo de um dado oscilador esteja estável, antes de permitir o envio do sinal de *clock* para a CPU.

- **Power-up Timer (PWRT):** é um contador de 11bits, que usa o INTRC como fonte de *clock*, o qual fornece um atraso aproximadamente igual a 65.6ms, proveniente de $2^{11} * 32\mu\text{s} = 65.536\mu\text{s}$. Esse contador é habilitado, limpando o bit de configuração PWRTEN (`#pragma config PWRT = ON`).
- **Oscillator Start-up Timer (OST):** tem a intenção de garantir que o sinal de *clock* esteja estável, antes que o oscilador envie sinal de *clock* para o dispositivo. O OST gera um atraso equivalente a 1024 ciclos de máquina (Tosc), logo após o PWRT, e só funciona no modo XT, HS e HSPLL, e no *Power-on Reset* ou quando o dispositivo sai do modo de gerenciamento de energia (*Sleep* e *Idle*). O bit de status associado a esse contador é OSTS (`OSCCON<3>`). Se for '0', indica que o oscilador primário ainda não está pronto para fornecer sinal de *clock*; caso contrário, significa que o OST expirou e o oscilador primário está rodando.
- **PLL Lock Time-out:** quando o PLL está habilitado, um contador especial é utilizado para prover um tempo de atraso fixo de aproximadamente 2ms (TPLL=2ms).

Tabela 5.6: Várias situações do *time-out*.

Configuração do oscilador	Power-up e Brown-out		Saída do modo de gerenciamento de energia
	PWRT = ON	PWRT = OFF	
HS, XT	66ms + 1024Tosc	1024Tosc	1024Tosc
HSPLL, XTPLL	66ms + 1024Tosc + 2ms	1024Tosc + 2ms	1024Tosc + 2ms
EC, ECIO	66ms	-	-
ECPLL, ECPIO	66ms + 2ms	2ms	2ms
INTIO, INTCKO	66ms	-	-
INTHS, INTXT	66ms + 1024Tosc	1024Tosc	1024Tosc

5.8.3 Two-Speed Start-Up

A função *Two-Speed Start-up* é habilitada pelo bit de configuração IESO (#pragma config IESO = ON) e a sua principal função é selecionar o oscilador interno INTRC como fonte de *clock* até que o *clock* primário esteja disponível, ou seja, complete o atraso de *start-up*.



Essa função deve ser utilizada somente no modo XT, HS, XTPLL ou HSPLL.

5.9 Características Elétricas do PIC18F4550

Faixa de temperatura de trabalho	-40°C a +85°C
Voltagem de funcionamento	4V a 5.5V
Voltagem máxima no pino V_{DD} em relação a V_{SS}	-0.3V a +7.5V
Potência máxima de dissipação	1W
Corrente máxima de saída do pino V_{SS}	300mA
Corrente máxima de entrada no pino V_{DD}	250mA
Corrente máxima fornecida por qualquer pino I/O	25mA
Corrente máxima de entrada em todas as portas I/O	200mA
Corrente máxima fornecida por todas as portas I/O.....	200mA

5.10 Fonte de Alimentação

A alimentação do circuito é essencial para o bom funcionamento do microcontrolador. Se for mal projetada, pode introduzir ruídos indesejáveis no sistema e reiniciar o dispositivo.

Veja a seguir um exemplo que utiliza um regulador de tensão de 5V (LM7805) de baixa potência, capaz de fornecer uma corrente de saída de no **máximo** 1A.

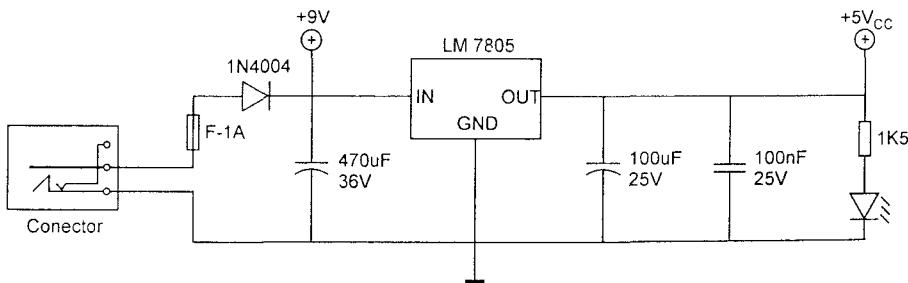


Figura 5.18: Alimentação.

Para maior confiabilidade do sistema, é recomendável a introdução de capacitores de desacoplamento (cerâmico - código 104) de capacidade equivalente a 100nF, o mais próximo possível dos pinos de alimentação do microcontrolador, a fim de estabilizar a tensão de alimentação e reduzir os ruídos que podem eventualmente afetar o desempenho do microcontrolador. A Figura 5.19 mostra um exemplo de como devem ser postos os capacitores de desacoplamento.

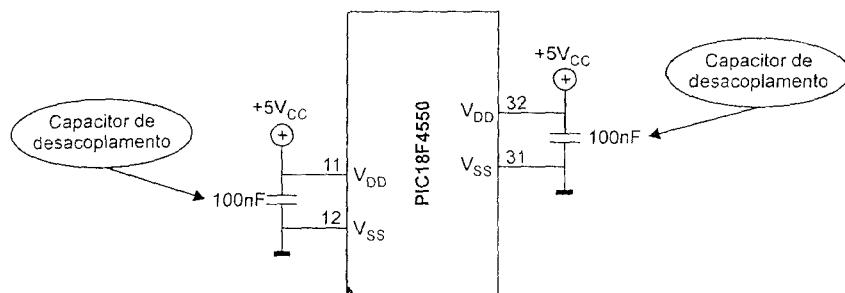


Figura 5.19: Conexão dos capacitores de desacoplamento

O microcontrolador PIC® também possui uma entrada de alimentação dedicada ao conversor A/D (V_{CC} e V_{SS}). Embora seja opcional, é recomendável separar a fonte de alimentação analógica da fonte digital, com o objetivo de reduzir o ruído gerado pela fonte digital (V_{CC} e V_{SS}) e garantir melhor precisão na conversão do sinal analógico.

5.11 Frequência x Tensão de Alimentação

Os microcontroladores PIC18F4550 podem ser adquiridos com a faixa de tensão padrão ou estendida. Porém, a tensão de alimentação está diretamente relacionada à frequência de trabalho do dispositivo. Vejamos na Figura 5.20 a relação entre frequência e tensão de alimentação.

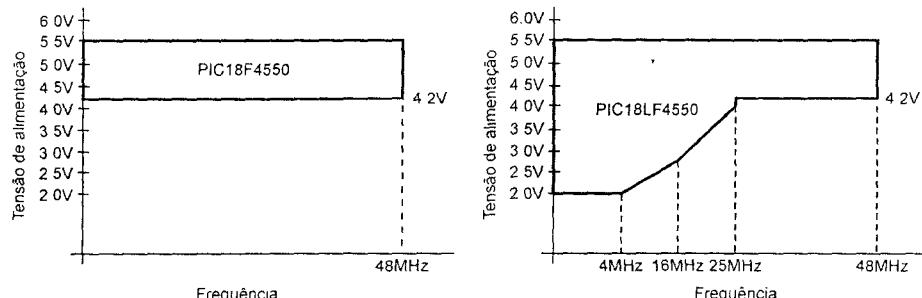


Figura 5.20: Frequência x Tensão de alimentação.

5.12 Funções Diversas do PIC18

Veremos neste tópico alguns conceitos pertinentes ao modo de funcionamento de algumas funções dos microcontroladores PIC18.

5.12.1 Registrador de Status

O *Status Register (STATUS)* contém os status das operações aritméticas realizadas pela ULA (Unidade Lógica Aritmética). Os bits relacionados a esse registrador são:

Tabela 5.7. Voltagem do Brown-out Reset (BOR).

STATUS	Nome do bit	Descrição
STATUS<0>	C - Carry Flag	Indica se o bit mais significativo do resultado foi transportado para fora 1: o bit mais significativo do resultado foi transportado para fora 0: o bit mais significativo do resultado não foi transportado para fora
STATUS<1>	DC - Digit Carry	Indica se o quarto bit menos significativo foi transportado para fora 1: o quarto bit menos significativo foi transportado para fora. 0: o quarto bit menos significativo não foi transportado para fora
STATUS<2>	Z - Zero Flag	Informa se a operação lógica ou aritmética resultou em um valor igual a zero. 1: o resultado da operação lógica ou aritmética foi igual a zero 0: o resultado da operação lógica ou aritmética foi diferente de zero.
STATUS<3>	OV - Overflow	Esse bit é usado em operações aritméticas sinalizadas. Ele indica o transbordamento do sétimo bit, o qual modifica o estado do bit de sinal. 1: ocorreu transbordamento. 0: não ocorreu transbordamento.
STATUS<4>	N - Negative	Mostra o sinal do resultado da operação aritmética. 1: negativo 0: positivo.

5.12.2 Fall-Safe Clock Monitor (FCM)

Monitora a fonte de *clock*. Se ocorrer algum tipo de falha no oscilador externo, o dispositivo continua operando, pois ao verificar a falha, o controlador gera uma interrupção e modifica a fonte de *clock* do dispositivo para o bloco do oscilador interno INTRC e reinicia o *Watchdog Timer* (WDT). Para habilitar essa opção, setar o bit FCMEN (`#pragma config FCMEN = ON`).

5.12.3 Instruções Estendidas

As instruções estendidas enviam comandos de duas palavras e quando habilitadas pelo bit de configuração XINST (`#pragma config XINST = ON`), passam a ser válidas no PIC18. As instruções são ADDFSR, ADDULNK, CALLW, MOVSS, PUSHL, SUBFSR e SUBULNK.

5.12.4 High/Low-Voltage Detect (HLVD)

O módulo de detecção de alta/baixa voltagem é habilitado pelo bit HLVDEN (HLVDCON<4> = 1) e permite verificar se a tensão de alimentação do dispositivo (V_{cc}) ou uma tensão externa, presente no pino definido como HLVDIN, é maior ou menor que uma determinada tensão de referência. O bit de configuração da condição de disparo desse evento é o VDIRMAG (HLVDCON<7>).

Para VDIRMAG=1, o evento é disparado se a tensão na entrada HLVDIN \geq tensão de referência (HLVLD3:HLVLD0).

Para VDIRMAG=0, o evento é disparado se a tensão na entrada HLVDIN \leq tensão de referência (HLVLD3:HLVLD0).

Toda vez que o módulo HLVD é habilitado (HLVDEN=1), o circuito relacionado a ele requer um determinado tempo para se estabilizar. Quando bit IRVST (HLVDCON<5>) é setado e o módulo está pronto para gerar interrupção.

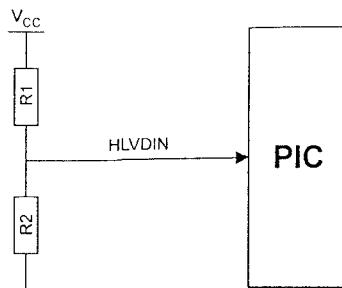


Figura 5.21. Circuito externo para verificação da tensão de alimentação V_{CC}.

A tensão de referência para esse módulo pode ser configurada por meio dos bits HLVDL3:HLVDL0 (HLVDCON<3:0>), cuja combinação permite obter as seguintes faixas de tensão:

Tabela 5.8: Faixas de tensão de referência.

HLVDL3:HLVDL0	Tensão típica (V)	HLVDL3:HLVDL0	Tensão típica (V)
0000	2.17	1000	3.39
0001	2.23	1001	3.55
0010	2.36	1010	3.71
0011	2.44	1011	3.90
0100	2.60	1100	4.11
0101	2.79	1101	4.33
0110	2.89	1110	4.59
0111	3.12		

No instante em que esse evento é disparado, uma interrupção pode ser gerada se o bit HLVDE (PIE2<2> = 1) for igual a 1 e sinalizada pelo bit HLVDIF (PIR2<2>). Sendo HLVDIF=1, indica a ocorrência da condição de interrupção e deve ser limpo manualmente, sempre que a interrupção ocorrer.

5.13 Métodos de Programação

Existem basicamente três maneiras de efetuar a gravação do programa no microcontrolador PIC®. O primeiro modo é retirar o microcontrolador do circuito e conectá-lo a um suporte especial do gravador. O segundo é o modo ICSP® (*In Circuit Serial Program*) que permite a gravação do código, sem a necessidade de retirar o microcontrolador do circuito, bastando a utilização de quatro pinos (V_{pp}, V_{ss}, PGC, PGD). O terceiro é o modo LVP (*Low Voltage Program*) que é semelhante ao modo ICSP®, mas não utiliza uma alta tensão (12 volts) no pino V_{pp}, em contrapartida necessita que um pino I/O seja reservado exclusivamente para a função de gravação.



Muitos microcontroladores são danificados devido à inversão dos pinos na hora da gravação. Por este motivo é recomendável adotar um tipo de conector padrão a fim de evitar prejuízos em suas experiências.

5.14 Tipos de Encapsulamento

A Tabela 5.9 lista os principais tipos de encapsulamento de microcontroladores PIC®.

Tabela 5.9: Tipos de encapsulamento.

Encapsulamento	Exemplo	Imagem
PDIP	PIC18F2550 - I/SP PIC18F4550 - I/P	
TQFP	PIC18F4550 - I/PT	
PLCC	PIC18F4431 - I/L	
QFN	PIC18F4580 - I/ML	
SOIC	PIC18F2550 - I/SO	
SSOP	PIC18F4550 - I/SS	

5.15 Identificação do Microcontrolador PIC

Tabela 5.10: Identificação do microcontrolador PIC® com memória FLASH.

PART NO.	X	IXX	XXX
Dispositivo	Range de temperatura	Encapsulamento	Estampa
Dispositivo:	PIC18F4550, PIC18F4580, PIC18F4431, PIC16C925 PIC18LF4550, PIC18LF4580, PIC18LF4431, PIC16LC925		
Nota.	F = Memória FLASH com range de tensão padrão (4,0 a 5,5V) C = Memória OTP/EPROM com range de tensão padrão (4,0 a 5,5V) CR = Memória ROM com range de tensão padrão (4,0 a 5,5V) LF = Memória FLASH com range de tensão estendido (2,0 a 5,5V) LC = Memória OTP/EPROM com range de tensão estendido (2,0 a 5,5V)		
Range de temperatura.	I = -40 a 80°C (Industrial)		
Encapsulamento:	ML = QFN PT = TQFP SO = SOIC SP = Skinny Plastic DIP P = PDIP L = PLCC SS = SSOP JW = Janelado (Memória EPROM)		
Estampa:	QTP, SQTP, Código ou requerimentos especiais		

Exemplo

1. PIC18F4550 - I/P = temperatura industrial, encapsulamento PDIP, faixa de tensão padrão.
2. PIC18LF4550 - I/SO = temperatura industrial, encapsulamento SOIC, faixa de tensão estendida.

5.16 Arquivo de Cabeçalho

Todos os modelos de microcontrolador PIC18 suportados pelo compilador MPLAB® C18 possuem uma biblioteca com informações sobre o modelo e definições utilizadas pelo compilador. Essas bibliotecas, também conhecidas como arquivos de cabeçalho, são obrigatórias em todos os códigos de programa, e estão localizadas na pasta "C:\MCC18\h".

Praticamente todos os registros do dispositivo estão implementados no arquivo de cabeçalho, e podem ser acessados através de uma das seguintes sintaxes:

Sintaxe

```
valor_8bits = <nome_registro>
<nome_registro> = valor_8bits
valor_1bit = <nome_registro>bits.<nome_bit>
<nome_registro>bits.<nome_bit> = valor_1bit
```

Sendo:

- **nome_registro**: nome do registro.
- **nome_bit**: nome do *bit* associado ao registro.
- **valor_8bits**: valor de *8bits*.
- **valor_1bit**: valor 0 ou 1.

A diferença entre as sintaxes é fácil de ser observada. Quando somente o nome do registrador é mencionado, o comando de leitura/escrita é feito nos *8bits* do registro. O acesso de apenas um determinado *bit* do registro é feito através de estruturas declaradas no arquivo de cabeçalho, sendo o valor de escrita/leitura sempre 0 ou 1.

Exemplo

```
// Realiza uma operação de escrita no registro PORTB.
// Coloca o nível de tensão igual a 0V em todos os pinos da porta B configurados como saída.
PORTB = 0b00000000;
```

```
// Realiza uma operação de leitura do registro TMR1L.
// Lê o registro TMR1L e armazena o seu valor na variável contador_TIMER1.
contador_TIMER1 = TMR1L;
```

```
// Realiza uma operação de escrita em apenas um bit do registro RCON.
// Seta o bit IPEN do registro RCON.
// Habilita interrupção com nível de prioridade.
RCONbits.IPEN = 1;
```

```
// Realiza uma operação de leitura de somente um bit do registro INTCON.
// Lê o bit INT0IF e armazena o valor na variável checa_INT0.
checa_INT0 = INTCONbits.INT0IF;
```

Exercícios

1. Qual a descrição correta do modelo PIC18LF4550 - IISO?
 - a) Temperatura industrial, encapsulamento PDIP, faixa de tensão padrão.
 - b) Temperatura industrial, encapsulamento PLCC, faixa de tensão padrão.
 - c) Temperatura industrial, encapsulamento SOIC, faixa de tensão padrão.
 - d) Temperatura industrial, encapsulamento SOIC, faixa de tensão estendida.
 - e) N.d.a.
2. Como deve ser feita a ligação entre o gravador e o PIC18F4550 para efetuar uma gravação ICSP? Consulte a Figura 5.2.

Pinos do gravador	Pinos do PIC18F4550
V _{PP}	Pino 12 ou 31
PGD	Pino 39
PGC	Pino 40
V _{SS}	Pino 1

3. Qual é a função dos capacitores de desacoplamento?
 - I. Estabilizar a tensão de alimentação.
 - II. Aumentar a tensão de alimentação.
 - III. Filtrar ruídos provenientes do sistema.
 - IV. Não há utilidade.

Entre as alternativas listadas anteriormente, assinale a afirmativa correta:

 - a) As alternativas II e III estão corretas.
 - b) As alternativas I e III estão corretas.
 - c) A alternativa IV está correta.
 - d) Somente a alternativa I está correta.
4. Quais são os benefícios de utilizar um conector-padrão para efetuar uma gravação ICSP®?
 - I. Evitar inversão dos pinos.
 - II. Diminuir as chances de danificar o dispositivo.
 - III. Aumentar as chances de danificar o dispositivo.
 - IV. Aumentar a velocidade de gravação.
 - V. Agilizar a conexão entre o gravador e o microcontrolador.

Entre as alternativas listadas anteriormente, assinale a afirmativa correta:

 - a) As alternativas I e IV estão corretas.
 - b) As alternativas I, II e IV estão corretas.
 - c) Somente a alternativa IV está correta.
 - d) As alternativas I, II e V estão corretas.
 - e) N.d.a.
5. Suponha que um cristal de 10MHz esteja sendo usado para fornecer sinal de *clock* para o microcontrolador. Isso significa que um ciclo de máquina equivale a:

a) 10MHz.	d) 2.5MHz.
b) 5MHz.	e) 3MHz.
c) 4MHz.	

6

Configuração do PIC18

Existem duas maneiras de configurar o microcontrolador PIC18. A primeira é pelo software MPLAB® IDE, que já foi explicado no Capítulo 2, e o segundo modo pode utilizar a diretiva `#pragma config`. Com essa diretiva é possível configurar o tipo de oscilador, *Watchdog Timer*, proteção de código, *Power up Timer*, *Master Clear* etc.



Essas configurações podem contar com o auxílio de uma ou mais diretivas `#pragma config`

Sintaxe

```
#pragma config lista_configuração=valor_nome
```

Sendo

- **lista_configuração**: lista das configurações do dispositivo.
- **valor_nome**: nome ou valor atribuído a configuração em questão.

Ambos os parâmetros são específicos a cada dispositivo. O modo de utilização de cada parâmetro pode ser visualizado, executando o comando "mcc18 --help-config" no **command-line**, localizado na aba MPLAB C18 em **Project → Build Options → Project**. A Tabela 6.1 lista as configurações disponíveis no PIC18F4550.

Tabela 6.1: Lista das constantes relacionadas a configurações do PIC18F4550.

Oscilador		
lista_configuração	valor_nome	Descrição
FOSC	XT_XT	Oscilador XT, XT usado pelo USB
	XTPLL_XT	Oscilador XT com PLL habilitado, XT usado pelo USB
	ECIO_EC	Clock externo, pino RA6 configurado como I/O, EC usado pelo USB
	EC_EC	Clock externo, CLKOUT (Fosc/4) sobre o pino RA6, EC usado pelo USB
	ECPLLIO_EC	Clock externo com PLL habilitado, pino RA6 configurado como I/O, EC usado pelo USB
	ECPLL_EC	Clock externo com PLL habilitado, CLKOUT (Fosc/4) sobre o pino RA6, EC usado pelo USB
	INTOSCIO_EC	Oscilador interno, pino RA6 configurado como I/O, EC usado pelo USB
	INTOSC_EC	Oscilador interno, CLKO (Fosc/4) sobre o pino RA6, EC usado pelo USB
	INTOSC_XT	Oscilador interno, XT usado pelo USB
	INTOSC_HS	Oscilador interno, HS usado pelo USB
	HS	Oscilador HS, HS usado pelo USB
	HSPLL_HS	Oscilador HS com PLL habilitado, HS usado pelo USB.

Prescale (PLL)		
lista_configuração	valor_nome	Descrição
PLLDIV	1	Oscilador de entrada equivalente a 4MHz sem prescale
	2	Oscilador de entrada equivalente a 8MHz dividido por 2
	3	Oscilador de entrada equivalente a 12MHz dividido por 3
	4	Oscilador de entrada equivalente a 16MHz dividido por 4
	5	Oscilador de entrada equivalente a 20MHz dividido por 5
	6	Oscilador de entrada equivalente a 24MHz dividido por 6
	10	Oscilador de entrada equivalente a 40MHz dividido por 10
	12	Oscilador de entrada equivalente a 48MHz dividido por 12
	Divisor de Clock do Sistema	
lista_configuração	valor_nome	Descrição
CPUDIV	OSC1_PLL2	Sem PLL - OSC1/OSC2 dividido por 1 Com PLL - 96 MHz PLL dividido por 2
	OSC2_PLL3	Sem PLL - OSC1/OSC2 dividido por 2 Com PLL - 96 MHz PLL dividido por 3
	OSC3_PLL4	Sem PLL - OSC1/OSC2 dividido por 3 Com PLL - 96 MHz PLL dividido por 4
	OSC4_PLL6	Sem PLL - OSC1/OSC2 dividido por 4 Com PLL - 96 MHz PLL dividido por 6
Clock da USB - Usado somente no modo Full Speed (UCFG:FSEN = 1)		
lista_configuração	valor_nome	Descrição
USBDIV	1	A fonte de clock para a USB vem diretamente do oscilador primário
	2	A fonte de clock para a USB vem do bloco 96 MHz PLL dividido por 2
Fail-Safe Clock Monitor		
lista_configuração	valor_nome	Descrição
FCMEN	OFF	Desabilita o monitoramento do sinal de clock
	ON	Habilita o monitoramento do sinal de clock
Mudança do oscilador interno/externo		
lista_configuração	valor_nome	Descrição
IESO	OFF	Desabilita a mudança do oscilador interno/externo
	ON	Habilita a mudança do oscilador interno/externo
Power-up Timer (PWRT)		
lista_configuração	valor_nome	Descrição
PWRT	ON	Habilita o Power-up Timer
	OFF	Desabilita o Power-up Timer
Brown-out Reset (BOR)		
lista_configuração	valor_nome	Descrição
BOR	OFF	Brown-out Reset desabilitado, tanto no hardware como no software
	SOFT	Brown-out Reset habilitado e controlado por software (SBOREN = 1)
	ON_ACTIVE	Brown-out Reset habilitado somente no hardware e desabilitado no modo Sleep (SBOREN = 0)
	ON	Brown-out Reset habilitado somente no hardware (SBOREN = 0)

Voltagem do Brown-out Reset (BOR)		
lista_configuração	valor_nome	Descrição
BORV	0	Aproximadamente 4,59V
	1	Aproximadamente 4,33V
	2	Aproximadamente 2,79V
	3	Aproximadamente 2,05V
Regulador interno de 3.3V (V_{usb})		
lista_configuração	valor_nome	Descrição
VREGEN	OFF	Desabilita o regulador de tensão da USB
	ON	Habilita o regulador de tensão da USB
Watchdog Timer (WDT)		
lista_configuração	valor_nome	Descrição
WDT	OFF	WDT desabilitado no hardware e controlado por software
	ON	WDT habilitado no hardware e desabilitado no software
Watchdog Timer (WDT) Postscale		
lista_configuração	valor_nome	Descrição
WDTPS	1	1:1
	2	1:2
	4	1:4
	8	1:8
	16	1:16
	32	1:32
	64	1:64
	128	1:128
	256	1:256
	512	1:512
	1024	1:1024
	2048	1:2048
	4096	1:4096
	8192	1:8192
	16384	1:16384
	32768	1:32768
	32768	1:32768
Master Clear Reset (MCLR)		
lista_configuração	valor_nome	Descrição
MCLRE	OFF	Habilita o pino RE3 como entrada e desabilita o MCLR
	ON	Desabilita o pino RE3 como entrada e habilita o MCLR
Oscilador Low-Power no TIMER1		
lista_configuração	valor_nome	Descrição
LPT1OSC	OFF	Timer1 configurado para operar no modo de alto consumo
	ON	Timer1 configurado para operar no modo de baixo consumo
Conversores A/D da porta B		
lista_configuração	valor_nome	Descrição
PBADEN	OFF	Configura os pinos PORTB<4:0> (RB0, RB1, RB2, RB3, RB4) como I/O sobre o Reset
	ON	Configura os pinos PORTB<4:0> (AN12, AN10, AN8, AN9, AN11) como canal de entrada de sinal analógico sobre o Reset

Multiplexação do pino do módulo CCP2		
lista_configuração	valor_nome	Descrição
CCP2MX	OFF	Defino o pino RB3 como sendo a I/O utilizada pelo módulo CCP2
	ON	Defino o pino RC1 como sendo a I/O utilizada pelo módulo CCP2
Stack Full/Underflow Reset		
lista_configuração	valor_nome	Descrição
STVREN	OFF	Desabilita o Reset por Stack full/underflow
	ON	Habilita o Reset por Stack full/underflow
Single-Supply ICSP		
lista_configuração	valor_nome	Descrição
LVP	OFF	Desabilita o Single-Supply ICSP
	ON	Habilita o Single-Supply ICSP
Porta dedicada à programação/debugação In-Circuit. (ICPORT)		
lista_configuração	valor_nome	Descrição
ICPRT	OFF	ICPORT desabilitado
	ON	ICPORT habilitado
Instruções estendidas		
lista_configuração	valor_nome	Descrição
XINST	OFF	Repertório de instruções estendido e modo de endereçamento indexado desabilitado (modo legal)
	ON	Repertório de instruções estendido e modo de endereçamento indexado habilitado
In-circuit Debugger		
lista_configuração	valor_nome	Descrição
DEBUG	ON	Debugação habilitada e os pinos RB6 e RB7 dedicados ao In-Circuit Debug
	OFF	Debugação desabilitada e os pinos RB6 e RB7 configurados como I/O
Proteção do código no bloco 0		
lista_configuração	valor_nome	Descrição
CP0	ON	Código no bloco 0 (000800-001FFFh) protegido
	OFF	Código no bloco 0 (000800-001FFFh) desprotegido
Proteção do código no bloco 1		
lista_configuração	valor_nome	Descrição
CP1	ON	Código no bloco 1 (002000-003FFFh) protegido
	OFF	Código no bloco 1 (002000-003FFFh) desprotegido
Proteção do código no bloco 2		
lista_configuração	valor_nome	Descrição
CP2	ON	Código no bloco 2 (004000-005FFFh) protegido
	OFF	Código no bloco 2 (004000-005FFFh) desprotegido
Proteção do código no bloco 3		
lista_configuração	valor_nome	Descrição
CP3	ON	Código no bloco 3 (006000-007FFFh) protegido
	OFF	Código no bloco 3 (006000-007FFFh) desprotegido
Proteção do código no bloco boot		
lista_configuração	valor_nome	Descrição
CPB	ON	Código no bloco boot (000000-0007FFh) protegido
	OFF	Código no bloco boot (000000-0007FFh) desprotegido
Proteção do código no EEPROM		
lista_configuração	valor_nome	Descrição
CPD	ON	Código na EEPROM protegido.
	OFF	Código na EEPROM desprotegido

Proteção de escrita no bloco 0		
lista_configuração	valor_nome	Descrição
WRT0	ON	Proteção de escrita no bloco 0 (000800-001FFFFh) habilitada
	OFF	Proteção de escrita no bloco 0 (000800-001FFFFh) desabilitada
Proteção de escrita no bloco 1		
lista_configuração	valor_nome	Descrição
WRT1	ON	Proteção de escrita no bloco 1 (002000-003FFFFh) habilitada
	OFF	Proteção de escrita no bloco 1 (002000-003FFFFh) desabilitada
Proteção de escrita no bloco 2		
lista_configuração	valor_nome	Descrição
WRT2	ON	Proteção de escrita no bloco 2 (004000-005FFFFh) habilitada
	OFF	Proteção de escrita no bloco 2 (004000-005FFFFh) desabilitada
Proteção de escrita no bloco 3		
lista_configuração	valor_nome	Descrição
WRT3	ON	Proteção de escrita no bloco 3 (006000-007FFFFh) habilitada
	OFF	Proteção de escrita no bloco 3 (006000-007FFFFh) desabilitada
Proteção de escrita no bloco boot		
lista_configuração	valor_nome	Descrição
WRTB	ON	Proteção de escrita no bloco boot (000000-0007FFh) habilitada
	OFF	Proteção de escrita no bloco boot (000000-0007FFh) desabilitada
Proteção de escrita no registrador de configuração		
lista_configuração	valor_nome	Descrição
WRTRC	ON	Proteção de escrita no registrador de configuração (300000-3000FFh) habilitada
	OFF	Proteção de escrita no registrador de configuração (300000-3000FFh) desabilitada
Proteção de escrita na EEPROM		
lista_configuração	valor_nome	Descrição
WRTE	ON	Proteção de escrita na EEPROM habilitada
	OFF	Proteção de escrita na EEPROM desabilitada
Proteção de leitura da tabela no bloco 0		
lista_configuração	valor_nome	Descrição
EBTR0	ON	Bloco 0 (000800-001FFFFh) protegido contra o comando de leitura da tabela, executado a partir de outro bloco
	OFF	Bloco 0 (000800-001FFFFh) desprotegido contra o comando de leitura da tabela, executado a partir de outro bloco
Proteção de leitura da tabela no bloco 1		
lista_configuração	valor_nome	Descrição
EBTR1	ON	Bloco 1 (002000-003FFFFh) protegido contra o comando de leitura da tabela, executado a partir de outro bloco
	OFF	Bloco 1 (002000-003FFFFh) desprotegido contra o comando de leitura da tabela, executado a partir de outro bloco
Proteção de leitura da tabela no bloco 2		
lista_configuração	valor_nome	Descrição
EBTR2	ON	Bloco 2 (004000-005FFFFh) protegido contra o comando de leitura da tabela, executado a partir de outro bloco
	OFF	Bloco 2 (004000-005FFFFh) desprotegido contra o comando de leitura da tabela, executado a partir de outro bloco

Proteção de leitura da tabela no bloco 3		
lista_configuração	valor_nome	Descrição
EBTR3	ON	Bloco 3 (006000-007FFFh) protegido contra o comando de leitura da tabela, executado a partir de outro bloco
	OFF	Bloco 3 (006000-007FFFh) desprotegido contra o comando de leitura da tabela, executado a partir de outro bloco
Proteção de leitura da tabela no bloco boot		
lista_configuração	valor_nome	Descrição
EBTRB	ON	Bloco boot (000000-0007FFFh) protegido contra o comando de leitura da tabela, executado a partir de outro bloco
	OFF	Bloco boot (000000-0007FFFh) desprotegido contra o comando de leitura da tabela, executado a partir de outro bloco

Exemplo

```
#pragma config FOSC = HS // Oscilador empregado nos projetos. Fosc = 20MHz
#pragma config CPUDIV = OSC1_PLL2
#pragma config WDT = OFF           //Desabilita o Watchdog Timer (WDT).
#pragma config PWRT = ON          //Habilita o Power-up Timer (PWRT).
#pragma config BOR = ON            //Brown-out Reset (BOR) habilitado somente no hardware.
#pragma config BORV = 1             //Voltage do BOR é 4,33V.
#pragma config PBADEN = OFF        //RB0,1,2,3 e 4 configurado como I/O digital.
#pragma config LVP = OFF           //Desabilita o Low Voltage Program.
```



Todos os projetos propostos neste livro que utilizam esta mesma configuração incluem o arquivo **config_PIC18F4550.h**. Se o leitor preferir, pode configurar via MPLAB® IDE.

Veja a seguir a mesma configuração feita via MPLAB® IDE. (Capítulo 2)

Address	Value	Category	Setting
300000	00	96MHz PLL Prescaler	No Divide (4Nohz input) [OSC1/OSC2 3rci /1]/(96MHz PLL Src: /2)
300001	0C	CPU System Clock Postscaler	Clock src from OSC1/OSC2 AS: USB-4S
300002	0E	OSC1/OSC2 Clock Monitor Enable	Divided
300003	1E	Power Up Timer	Enabled
300004	00	Brown Out Detect	Enabled in hardware, SBCREN disabled
300005	81	Brown Out Voltage Regulator	4.2V
300006	00	USB Voltage Regulator	Disabled
300007	00	Watchdog Timer	Disabled-Controlled by SWDTEN bit
300008	00	Watchdog Postscaler	1:32/64
300009	00	CCF2 Mix	R21
30000A	00	PortB A/D Enable	PCRB<4..0> configured as digital I/O on RESET
30000B	00	Low Power Timer One enable	Disabled
30000C	00	Master Clear Enable	MCLR Enabled, RES3 Disabled
30000D	00	System Overvoltage Reset	Enabled
30000E	00	Low Voltage Protection	Disabled
30000F	00	Dedicated In-Circuit Serial Port (ICD/ICSP)	Disabled
300010	00	Extended Instruction Set Enable bit	Disabled
300011	00	Code Protect 00000-01FFF	Disabled
300012	00	Code Protect 02000-03FFF	Disabled
300013	00	Code Protect 04000-05FFF	Disabled
300014	00	Code Protect 06000-07FFF	Disabled
300015	00	Data FF Read Protect	Disabled
300016	00	Table Write Protect 00000-01FFF	Disabled
300017	00	Table Write Protect 02000-03FFF	Disabled
300018	00	Table Write Protect 04000-05FFF	Disabled
300019	00	Table Write Protect 06000-07FFF	Disabled
30001A	00	Code Table Write Protect	Disabled
30001B	00	Table Write Protect Boot	Disabled
30001C	00	Data FF Write Protect	Disabled
30001D	00	Table Read Protect 00000-01FFF	Disabled
30001E	00	Table Read Protect 02000-03FFF	Disabled
30001F	00	Table Read Protect 04000-05FFF	Disabled
300020	00	Table Read Protect 06000-07FFF	Disabled
300021	00	Table Read Protect Boot	Disabled

Figura 6.1: Bits de configuração.

7

Portas I/O Digitais

Portas I/O (input/output) digitais são portas de entrada/saída de dados, cujos valores alternam entre '0' e '1'. Por essas portas o microcontrolador é capaz de obter informações do mundo exterior, como também atuar diretamente no controle ON/OFF de dispositivos, como, por exemplo, ligar/desligar uma lâmpada, válvula, motor, entre outros.

O PIC18F4550 tem cinco portas disponíveis (A, B, C, D e E). Os pinos associados a elas são multiplexados com diferentes funções de periféricos. Geralmente, quando um determinado periférico é habilitado, o pino relacionado a ele deixa de ser de propósito geral, e passa a desempenhar a função que lhe é concedida pelo periférico.

Cada porta tem três registradores associados à configuração. São eles:

- **Registrador TRIS:** configura o sentido do fluxo de dados de uma determinada porta.
- **Registrador PORT:** escreve/lê o nível dos pinos associados à porta,
- **Registrador LAT:** armazena o valor do último comando de escrita.

Veja o diagrama de operação de uma porta I/O genérica, Figura 7.1.

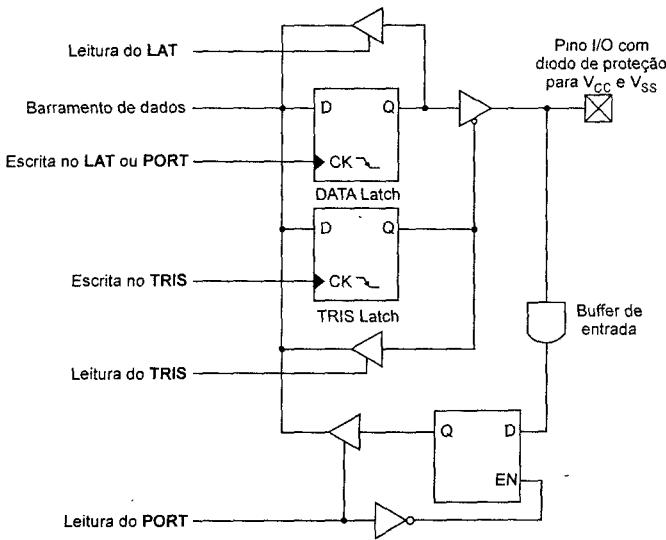


Figura 7.1: Diagrama de operação de uma porta I/O genérica.

Por meio do diagrama é possível verificar que o comando de escrita passa pelo *Data Latch* antes de realizar a modificação do estado de saída dos pinos.

O Registro *Data Latch* (**LAT**) também é uma memória mapeada e armazena o valor da última operação de escrita, no entanto essa operação não afeta os pinos I/O que estejam configurados como entrada ou sendo utilizados por um determinado periférico. O **LAT** retém o valor escrito e quando o pino for setado como saída de propósito geral, ele é afetado de acordo com o valor do **LAT**.

7.1 Sentido do Fluxo de Dados da Porta

O sentido do fluxo de dados de uma determinada porta é configurado pelo registrador **TRIS**. Ele tem 8bits, e cada elemento corresponde à configuração de um determinado pino de I/O. Sendo 0 - saída e 1 - entrada (modo de alta impedância).

O compilador MPLAB® C18 suporta comandos de acesso simultâneo dos 8bits do registrador **TRIS**, como também de um único *bit*.

7.1.1 TRISA, TRISB, TRISC, TRISD e TRISE

Os nomes dos identificadores de acesso aos 8bits dos registradores **TRIS** são os próprios nomes dos registros associados a eles.

Sintaxe

TRISx = valor

valor = **TRISx**

Sendo:

- **x**: nome da porta (letra maiúscula).
- **valor**: valor de 8bits. 0 (saída) ou 1 (entrada).

Suponha que desejamos configurar o fluxo de dados nos pinos da porta B do microcontrolador PIC18F4550. Logo, podemos enviar o seguinte comando:

TRISB = 0b00000001;

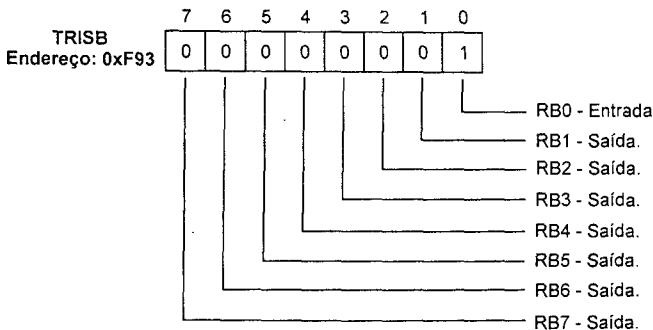


Figura 7.2: Configuração da porta B.

Exemplo

TRISA = 0b00011111; //RA0 a RA4 - entrada e RA5 a RA6 - saída.
TRISB = 0b11100000; //RB0,RB1,RB2,RB5,RB6 e RB7 - entrada e RB3 a RB4 - saída
TRISE = 0b00000000; //RE0 a RE2 - saída

7.1.2 TRISAbits, TRISBbits, TRISChits, TRISDbits e TRISEbits

Essas estruturas permitem o acesso de apenas um *bit* do registro **TRIS**.

Sintaxe

TRISXbits.TRISxy = *valor_bit*
valor_bit = **TRISXbits.TRISxy**

Sendo:

- **x**: nome da porta (letra maiúscula).
- **y**: número do pino.
- **valor_bit**: valor 0 (saída) ou 1 (entrada).

Exemplo

TRISAbits.TRISA5 = 1; //RA5 – entrada.
TRISBbits.TRISB7 = 0; //RB7 – saída.
TRISEbits.TRISE2 = 0; //RE2 – saída.

7.2 Controle do Estado dos Pinos da Porta

O status dos pinos de uma porta é armazenado no registrador **PORT**. Esse registrador possui um tamanho de 8bits e é responsável pelas operações de escrita e leitura dos pinos relacionados à porta. Sendo 0 – V_{ss} e 1 – V_{cc} .

7.2.1 PORTA, PORTB, PORTC, PORTD e PORTE

Os nomes dos identificadores de acesso aos 8bits dos registradores **PORT** são os próprios nomes dos registros associados a eles. Para um comando de leitura, o registrador **PORT** realiza a leitura do status dos pinos e para um comando de escrita, o *valor* é enviado para a porta *Latch* (**LAT**) que vai modificar os níveis dos pinos I/O configurados como saída.

Sintaxe

PORTx = *valor*
valor = **PORTx**

Sendo:

- **x**: nome da porta (letra maiúscula).
- **valor**: valor de 8bits.

Suponha que todos os pinos da porta B estejam configurados como saída (**TRISB** = 0x00), então podemos selecionar o nível de tensão na saída dos pinos da porta B do microcontrolador PIC18F4550, enviando o seguinte comando:

PORTB = 0b01010010;

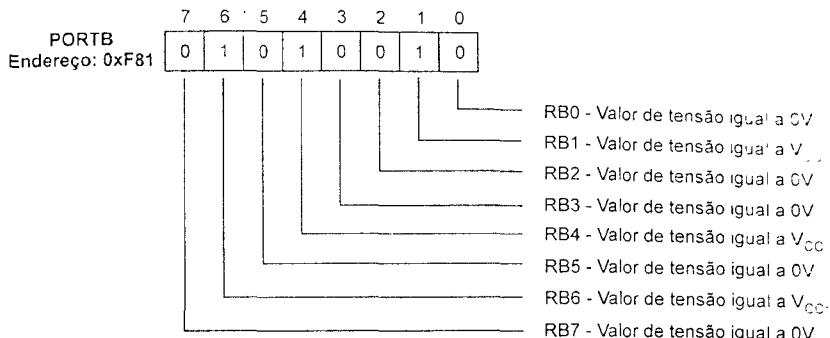


Figura 7.3 Controle da porta B.

Exemplo

TRISD = 0b00000000, //RD0 a RD7 – saída

TRISEbits.TRISE1 = 0; //RE1 – saída

PORTD = 0b01010010; //Seta os pinos da porta D, de acordo com a Figura 7.3.

PORTEbits RE1= ~ PORTEbits RE1; //Inverte o estado do pino RE1

7.2.2 PORTAbits, PORTBbits, PORTCbits, PORTDbits e PORTEbitsEssas estruturas permitem o acesso de apenas um *bit* do registro PORT.**Sintaxe**

```
PORTXbits.PORTxy = valor_bit
valor_bit = PORTXbits.PORTxy
```

Sendo:

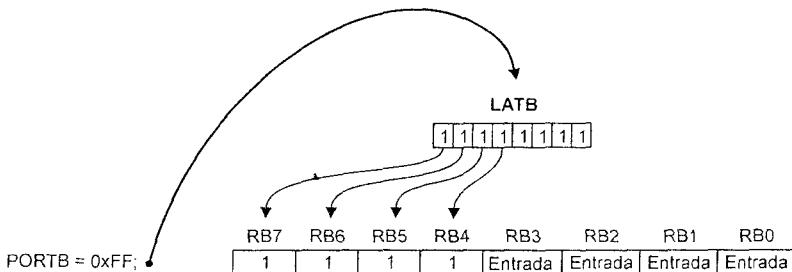
- **x**: nome da porta (letra maiúscula).
- **y**: nome do pino.
- **valor_bit**: valor 0 (V_{ss}) ou 1 (V_{cc}).

7.3 Registro LAT

Como visto anteriormente, o registro do *Data Latch* (LAT) armazena o valor enviado pelo comando de escrita. No entanto, o comando de escrita afeta somente os pinos configurados como saída de propósito geral, logo o LAT retém o valor do último comando de escrita e modifica o estado do pino assim que for configurado como saída. Veja na sequência um simples exemplo.

Considerando que todos os pinos da porta B estejam configurados como I/O de propósito geral, veja na Figura 7.4 como o registrador LAT se comporta com relação à sequência de instruções ilustradas.

RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
TRISB = 0b00001111;	Saída	Saída	Saída	Saída	Entrada	Entrada	Entrada



RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
TRISB = 0b00001100;	Saída	Saída	Saída	Saída	Entrada	Entrada	Saída

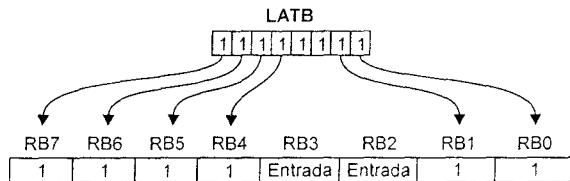


Figura 7.4: Comportamento do LAT.

7.3.1 LATA, LATB, LATC, LATD e LATE

Os 8bits do registro **LAT** são acessados por identificadores associados ao mesmo nome do registro. Para um comando de leitura o registrador **LAT** realiza a leitura do valor de saída do *Data Latch* e escreve o valor no registro **PORT**; para um comando de escrita, o *valor* é carregado para o *Data Latch* que vai modificar os níveis dos pinos I/O configurados como saída.

Sintaxe

LATx = valor
valor = **LATx**

Sendo:

- **x**: nome da porta (letra maiúscula).
- **valor**: valor de 8bits.

O exemplo a seguir executa as operações indicadas na Figura 7.4.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.  

#include<delays.h> //Adiciona a biblioteca de delay.  

#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)
```

```

void main(void)
{
    TRISD = 0b00001111; //RD0 a RD3 – entrada e RBD a RD7 – saída
    LATD = 0xFF; //Seta todas as portas configuradas como saída RD0 a RD3 não são afetados

    // Gera um atraso de 2 segundos Fosc=20MHz - Tmaq=4/Fosc=0,2us.
    Delay10KTCYx (200); //200*10.000*0,2us=0,4s
    Delay10KTCYx (200); // 0,4s
    // Fim do atraso de 2 segundos

    TRISD = 0b00001100; //Pinos RD2 e RD3 - entrada e o resto dos pinos da porta D - saída.

    // Gera um atraso de 2 segundos Fosc=20MHz - Tmaq=4/Fosc=0,2us.
    Delay10KTCYx (200); //200*10.000*0,2us=0,4s
    Delay10KTCYx (200); // 0,4s
    // Fim do atraso de 2 segundos

    LATDbits.LATD6 = 0; //Coloca a saída RD6 em nível baixo (0V)

    while(1); //Looping infinito
}

```

7.3.2 LATAbits, LATBbits, LATCbits, LATDbits e LATEbits

Essas estruturas permitem o acesso de apenas um *bit* do registro LAT.

Sintaxe

`LATxbits.LATxy = valor_bit`
`valor_bit = LATxbits.LATxy`

Sendo:

- **x**: nome da porta (letra maiúscula).
- **y**: nome do pino.
- **valor_bit**: valor 0 (V_{ss}) ou 1 (V_{cc}).

7.4 Habilita/Desabilita Pull-Ups Internos

No Capítulo 5 foi comentado que o microcontrolador PIC18F4550 possui *pull-ups* internos conectados nas portas B e D, e que podem ser habilitados quando os pinos estão configurados como entrada.

A corrente que passa pelo *pull-up* está na faixa de $50\mu A$ a $400\mu A$ para um $V_{cc} = 5V$ e tensão no pino igual a V_{ss} .

Os *pull-ups* da porta B são habilitados, limpando o bit RBPU (`INTCON2<7> = 0`).

Os *pull-ups* da porta D são habilitados, setando o bit RDPU (`PORTE<7> = 1`).



O **pull-up** é automaticamente desabilitado, quando o pino da porta está configurado como saída.

Exemplo

```

#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include<delays.h> //Adiciona a biblioteca de delay.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)
unsigned char cont;
void main(void)
{
    TRISB = 0b00000111; //RB0 a RB2 – entrada e RB3 a RB7 – saída.
    TRISD = 0b11110000; //RD0 a RD3 – saída e RD4 a RD7 – entrada.

    INTCON2bits.NOT_RBPU = 0, //Habilita os pull-ups da porta B.
    PORTEbits.RDPU = 1; //Habilita os pull-ups da porta D.

    for(cont = 0 ; cont < 10 ; cont++) { Delay10KTCYx (200); } // Gera um atraso de 4 segundos.

    INTCON2bits.NOT_RBPU = 1; //Desabilita os pull-ups da porta B.
    PORTEbits.RDPU = 0; //Desabilita os pull-ups da porta D.

    while(1); //Looping infinito.
}

```

Exercícios

- Qual é o registrador responsável pela configuração do sentido do fluxo de dados dos pinos?
 - PORT
 - LAT
 - RDPU
 - TRIS
 - N d.a.
 - Utilizando as funções de controle das saídas, o pino RB2 pode ser colocado em 1 (V_{cc}) utilizando as seguintes funções:
 - PORTBbits.RB2 = 0
 - PORTBbits.RB2 = 1
 - PORTB = 0b00000010
 - PORTB = 0b00000110Dentre as alternativas citadas anteriormente, assinale a afirmativa correta.
 - II e III estão corretas.
 - II e IV estão corretas.
 - I e III estão corretas.
 - I, III e IV estão corretas.
 - III e IV estão corretas.
 - Qual das opções seguintes define que os pinos RD1 e RD5 comportam-se como pinos de entrada e o resto dos pinos da porta D são de saída?
 - TRISD = 0b00010001
 - TRISD = 0b00100010
 - TRISD = 0b11101110
 - TRISD = 0b11011101
 - TRISD = 0b01100011
 - Qual das opções representa os endereços físicos dos registros TRISD, PORTD e LATD do PIC18F4550? (Veja o mapa dos SFRs no Capítulo 5)
 - 0x94, 0x84 e 0x8B
 - 0x95, 0xF3 e 0x8C
 - 0xF9D, 0xF8D e 0xF7D
 - 0xF95, 0xF83 e 0xF8C

7.5 Projeto

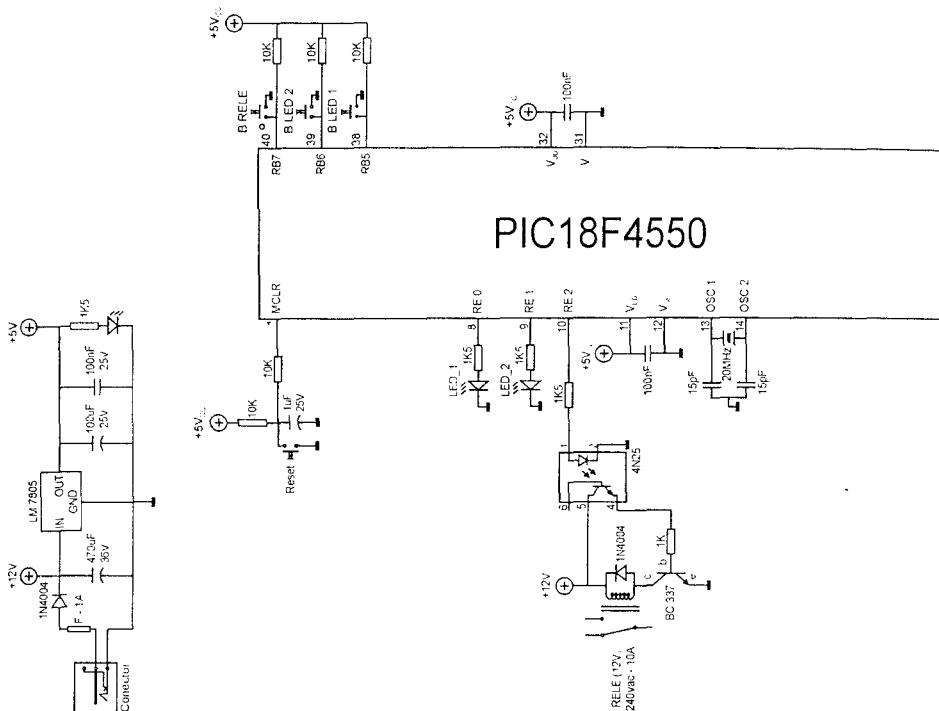


Figura 7.5. Circuito de acionamento do RELÉ, LED_1 e LED_2 pelo teclado.

Código

```
*****Projeto Capítulo 7 - TESTE DE PORTAS I/O*****
**
** Este programa permite ligar/desligar o LED_1 , LED_2 e o RELÉ através de três botões.
** Se o botão B_LED_1 é pressionado, então o LED_1 é ligado; caso contrário, é desligado.
** Se o botão B_LED_2 é pressionado, então o LED_2 é ligado; caso contrário, é desligado.
** Se o botão B_RELÉ é pressionado, então o RELÉ é ligado; caso contrário, é desligado
**
** Autor: Alberto Noboru Miyadaira
*****/
```

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <adc.h>           //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)

#define B_LED_1 PORTBbits.RB5          //Define outro nome para a estrutura.
#define B_LED_2 PORTBbits.RB6          //Define outro nome para a estrutura.

#define B_RELÉ PORTBbits.RB7          //Define outro nome para a estrutura.
#define LED_1 PORTBbits.RE0            //Define outro nome para a estrutura.
#define LED_2 PORTBbits.RE1            //Define outro nome para a estrutura.
#define RELE PORTEbits.RE2             //Define outro nome para a estrutura.
```

```

void main(void)
{
    TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0b00000000; //RD0 a RD7 – saída.
    TRISE = 0b00000000; //RE0 a RE2 – saída.

    //Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital. (Capítulo 13)
    //Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
    OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h
    CloseADC (); //Requer a biblioteca adc.h

    while(1) //Looping infinito.
    {
        if (B_LED_1 == 0)//Verifica se o valor presente no pino RB5 é 0v. (botão B_LED_1 pressionado)
        { LED_1 = 1; } //Liga o LED_1.
        else
        { LED_1 = 0; } //Desliga o LED_1.

        if (B_LED_2 == 0)//Verifica se o valor presente no pino RB6 é 0v. (botão B_LED_2 pressionado)
        { LED_2 = 1; } //Liga o LED_2.
        else
        { LED_2 = 0; } //Desliga o LED_2.

        if (B_RELÉ == 0)//Verifica se o valor presente no pino RB7 é 0v. (botão B_RELÉ pressionado)
        { RELÉ = 1; } //Liga o RELÉ.
        else
        { RELÉ = 0; } //Desliga o RELÉ.
    }
}

```



Display LCD 2X16

O *display* LCD alfanumérico é uma interface de saída capaz de mostrar caracteres do código ASCII, símbolos, além de alguns caracteres/símbolos especiais criados pelo próprio usuário. A interface entre o microcontrolador e o LCD alfanumérico é bastante simplificada, uma vez que ele possui um controlador interno capaz de reconhecer um conjunto de instruções predefinidas, transmitidas por meio de uma comunicação de quatro ou oito linhas de dados e três linhas de controle (RS, R/W e E).

O *display* LCD 2x16 (duas linhas e dezesseis colunas) foi adotado por esta obra, pois é um dispositivo bastante disseminado, no entanto existem outros modelos, que podem ser encontrados nas configurações 1x16, 2x16, 4x16, 2x20, 4x20, 2x40, 4x40 etc.

8.1 Pinagem LCD 2x16

As funções dos pinos de um *display* LCD alfanumérico são similares à maioria dos *displays* comercializados, porém nem todos apresentam a mesma disposição dos pinos. Antes de instalar o *display* LCD no circuito eletrônico, verifique a posição e descrição dos pinos no manual do fabricante, também chamado de *datasheet*.

A Figura 8.1 ilustra a pinagem do *display* LCD 2x16 utilizado nesta obra.

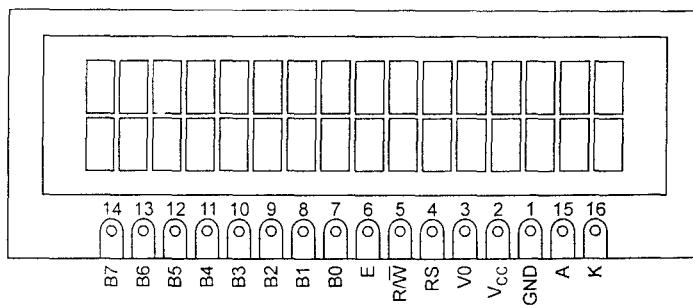


Figura 8.1: *Display* 2x16.

Tabela 8.1: Descrição dos pinos do display LCD 2x16.

Pino	Função	Descrição
1	GND	Terra
2	V _{CC}	Fonte de tensão. (5 V _{CC})
3	V _O	Contraste.
4	RS	0 - Instrução. 1 - Dado.
5	R/W	0 - Escrita. 1 - Leitura.
6	E	0 - Desabilita. 1 - Habilita.
7-14	B0-B7	Barramento de dados.
15	A	Anodo do LED de <i>backlight</i> (luz de fundo).
16	K	Catodo do LED de <i>backlight</i> (luz de fundo).

8.2 Instruções de Controle

Como estudado anteriormente, o *display* LCD possui um controlador interno capaz de reconhecer um conjunto de instruções predefinidas, que serão comentadas nesta seção.

8.2.1 Configuração do Cursor e do Display

As instruções relacionadas à configuração do *display* LCD estão listadas na Tabela 8.2.

Tabela 8.2: Configuração do cursor e *display*.

	Descrição	RS	R/W	Valor em hexadecimal	Delay
Cursor	Desloca o cursor para a esquerda quando um caractere é inserido, mas não desloca a mensagem.	0	0	0x04	40µs
	Desloca o cursor para a direita quando um caractere é inserido, mas não desloca a mensagem.	0	0	0x06	40µs
	Desloca o cursor e a mensagem para a esquerda quando um caractere é inserido.	0	0	0x05	40µs
	Desloca o cursor e a mensagem para a direita quando um caractere é inserido.	0	0	0x07	40µs
Display	Comunicação de 4bits. <i>Display</i> de uma linha. Matriz de 8x5.	0	0	0x20	40µs
	Comunicação de 4bits. <i>Display</i> de uma linha. Matriz de 10x5.	0	0	0x24	40µs
	Comunicação de 4bits. <i>Display</i> com duas ou mais linhas. Matriz de 8x5.	0	0	0x28	40µs
	Comunicação de 4bits. <i>Display</i> de duas ou mais linhas. Matriz de 10x5.	0	0	0x2C	40µs

	Descrição	RS	R/W	Valor em hexadecimal	Delay
Display	Comunicação de 8bits. Display de uma linha Matriz de 8x5	0	0	0x30	40µs
	Comunicação de 8bits. Display de uma linha. Matriz de 10x5.	0	0	0x34	40µs
	Comunicação de 8bits. Display de duas ou mais linhas. Matriz de 8x5.	0	0	0x38	40µs
	Comunicação de 8bits. Display de duas ou mais linhas. Matriz de 10x5.	0	0	0x3C	40µs

8.2.2 Controle do Display/Cursor

Uma vez configurados, o *display* e o cursor podem ser controlados com o auxílio das instruções listadas em seguida:

Tabela 8.3. Controle do *display/cursor*

	Descrição	RS	R/W	Valor em hexadecimal	Delay
	Desliga o cursor	0	0	0x0C	40µs
	Desliga o <i>display</i> .	0	0	0x08	40µs
	Desloca o cursor para a linha 1 e coluna 1. Observação: Não apaga o conteúdo da DDRAM.	0	0	0x02	1.6ms
	Desloca o cursor para direita, sem deslocar a mensagem.	0	0	0x14	40µs
	Desloca o cursor para esquerda, sem deslocar a mensagem.	0	0	0x10	40µs
	Liga o cursor com alternância.	0	0	0x0F	40µs
	Liga o <i>display</i> e o cursor	0	0	0x0E	40µs
	Liga o <i>display</i> e o cursor piscante.	0	0	0x0D	40µs
	Limpa a tela do <i>display</i> e desloca o cursor para a linha 1 e coluna 1. Observação: Apaga o conteúdo da DDRAM.	0	0	0x01	1.6ms

Os dados armazenados em cada posição do *display* ficam na memória interna, chamada DDRAM, que é volátil do tipo RAM, cujo conteúdo é perdido sempre que a alimentação de tensão é interrompida.

8.2.3 Controle da Mensagem

Tabela 8.4: Controle da mensagem.

	Descrição	RS	R/W	Valor em hexadecimal	Delay
	Desloca a mensagem para a direita.	0	0	0x1C	40µs
	Desloca a mensagem para a esquerda.	0	0	0x18	40µs

8.2.4 Status e Posição do Contador de Endereço

Para consultar o status do controlador interno do *display*, basta setar os pinos de controle RS = 0 e R/W = 1 e ler os dados do barramento. Esse comando solicita o envio do status do controlador, como também o contador de endereço.

Tabela 8.5: Status e posição do contador.

Descrição	RS	R/W	Valor em hexadecimal	Delay
Faz a leitura do status (s) e do contador de endereço (xxx xxxx).	0	1	sXXX XXXX	40µs

Note que o contador de endereço é representado por sete bits. Para obter a posição do cursor, basta fazer um OU lógico entre 0b1000000 e o valor retornado, ignorando o valor do status.

$$\text{Pos}_{\text{cursor}} = 0b1000000 \mid (\text{valor}_{\text{retornado}} + 1)$$

O valor de $\text{Pos}_{\text{cursor}}$ pode ser interpretado a partir das Tabelas 8.8 e 8.9.

8.2.5 Leitura e Escrita de Dados

Tabela 8.6: Leitura e escrita de dados.

Descrição	RS	R/W	Valor em hexadecimal	Delay
Insera um dado na posição do cursor.	1	0	xxxx xxxx	45µs
Faz a leitura do dado presente na posição do cursor.	1	1	xxxx xxxx	45µs

8.2.6 Endereço da Linha x Coluna

O endereço da posição do *display* LCD alfanumérico pode ser comparado com uma matriz linha x coluna, sendo cada posição representada por um endereço predeterminado. Sendo assim, para selecionar uma determinada posição do *display*, os pinos RS e R/W devem ser iguais a zero e então, enviar o endereço correspondente à posição.

Tabela 8.7: Endereço da posição do LCD.

Descrição	RS	R/W	Valor em hexadecimal	Delay
Desloca o cursor para uma determinada posição.	0	0	1xxx xxxx	40µs

Os endereços relacionados a cada posição do display LCD 2x16 e 4x20 podem ser vistos nas tabelas seguintes:

Tabela 8.8: Endereço da posição do LCD 2x16, representada em hexadecimal.

LCD 2x16	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Linha 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
Linha 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF

Tabela 8.9: Endereço da posição do LCD 4x20, representada em hexadecimal

LCD 4x20	1	2	3	4	5	6	7	8	9	10
Linha 1	80	81	82	83	84	85	86	87	88	89
Linha 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
Linha 3	94	95	96	97	98	99	9A	9B	9C	9D
Linha 4	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD

LCD 4x20	11	12	13	14	15	16	17	18	19	20
Linha 1	8A	8B	8C	8D	8E	8F	90	91	92	93
Linha 2	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3
Linha 3	9E	9F	A0	A1	A2	A3	A4	A5	A6	A7
Linha 4	DE	DF	E0	E1	E2	E3	E4	E5	E6	E7

8.2.7 Caractere Especial

O *display* LCD alfanumérico também possui uma memória dedicada ao armazenamento de caracteres criados pelo usuário, chamada CGRAM, uma memória volátil, cujos "caracteres especiais" devem ser novamente criados sempre que a alimentação do *display* for desligada. Essa memória, na maioria das vezes, permite que o usuário crie **oito** "caracteres especiais". A localização desses caracteres pode ser vista na Tabela 8.11.

Um "caractere especial" de 8bits pode ser criado obedecendo aos seguintes passos: primeiramente deve-se enviar a instrução 0x40 para setar o endereço da CGRAM, enviar uma sequência de 8bytes de dados, que vai compor o novo caractere (somente os cinco primeiros bits de cada byte enviado correspondem aos pixels do caractere, pois a matriz é de 8x5 ou 10x5). Caso queira criar mais de um caractere, basta enviar mais uma sequência de 8bytes de dados. A Tabela 8.10 mostra os passos para a criação de caractere.

Tabela 8.10: Criacão de caractere.

8.3 Inicialização do Display LCD 2x16 com Oito Vias

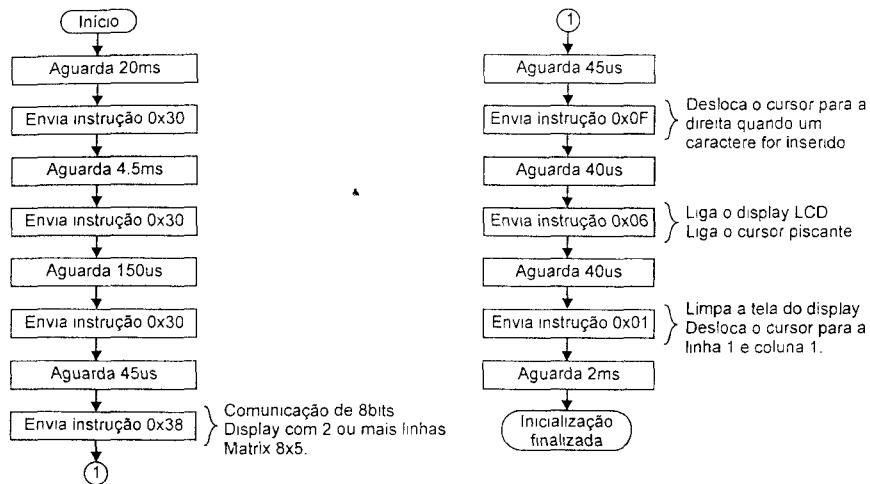


Figura 8.2: Fluxograma de inicialização do LCD 2x16 com oito vias.

8.4 Inicialização do Display LCD 2x16 com Quatro Vias

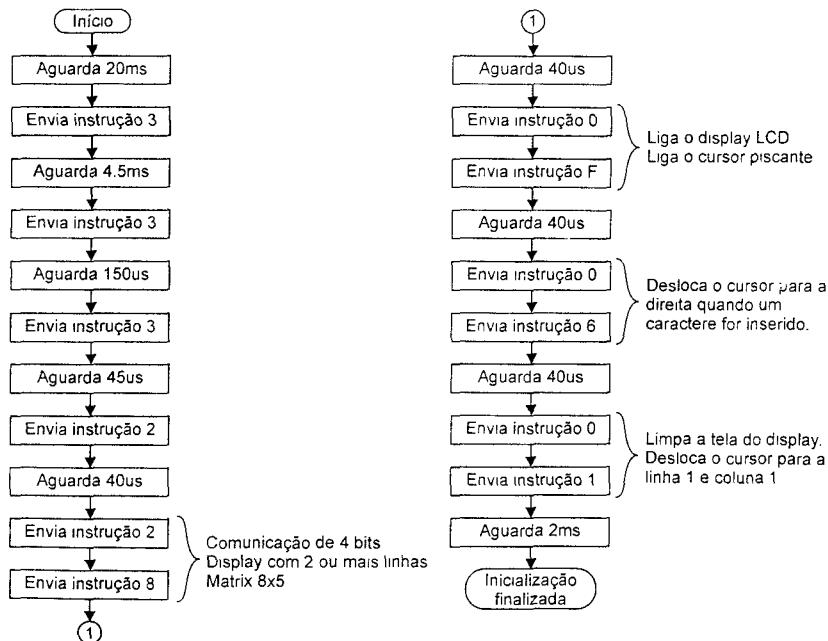


Figura 8.3: Fluxograma de inicialização do LCD 2x16 com quatro vias.

8.5 Conjuntos de Caracteres do Display

Tabela 8.11: Conjuntos de caracteres do display.

Upper 4 bit	Lower 4 bit	CG RAM (1)	CGP^pe33cep
LLLH	(2)		1A0a@	33333333
LLHL	(3)		2BRbr	44444444
LLHH	(4)		3CSScs	55555555
LHLL	(5)		4DTdt	66666666
LHLH	(6)		5SEUeu	77777777
LHHL	(7)		6SFUFU	88888888
LHHH	(8)		7BDJgJ	99999999
HLLL	(1)		8SHKhpx	AABBBBCC
HLLH	(2)		9IViV	BCCCCCCC
HLHL	(3)		JBjz	CCCCCCCC
HLHH	(4)		KDKk	CCCCCCCC
HHLL	(5)		L#I	CCCCCCCC
HHLH	(6)		Mm	CCCCCCCC
HHHL	(7)		Nn	CCCCCCCC
HHHH	(8)		Oo	CCCCCCCC

8.6 Biblioteca do Display LCD Alfanumérico

A biblioteca de manipulação do display LCD 2x16 (4 vias de comunicação) utilizado neste livro está disponível no site da Editora Érica (www.editoraerica.com.br). Vejamos na sequência a lista de comandos disponibilizados pelo arquivo.

Tabela 8.12: Descrição das funções contínuas na `biblioteca_lcd_2x16.h`.

Função	Descrição
<code>lcd_limpa_tela ()</code>	Limpa a tela do <i>display LCD</i>
<code>lcd_cursor_home ()</code>	Coloca o cursor na linha 1 e coluna 1 do <i>display</i> .
<code>lcd_desloca_cursor (direita_esquerda)</code>	Desloca o cursor para a direita ou esquerda 0 - Desloca o cursor para a direita. 1 - Desloca o cursor para a esquerda
<code>lcd_desloca_mensagem (direita_esquerda)</code>	Desloca a mensagem para a direita ou esquerda 0 - Desloca a mensagem para a direita. 1 - Desloca a mensagem para a esquerda
<code>lcd_LD_cursor (config)</code>	Liga/Desliga o cursor/ <i>display</i> . 0 - Desliga o cursor. 1 - Desliga o <i>display</i> . 2 - Liga o cursor com alternância 3 - Liga o <i>display</i> e o cursor. 4 - Liga o <i>display</i> e o cursor piscante.
<code>lcd_posicao (linha, coluna)</code>	Coloca o cursor em uma determinada posição do <i>display LCD</i> .
<code>lcd_escreve_dado (dado)</code>	Escreve um caractere ou símbolo no <i>display</i> .
<code>lcd_le_dado ()</code>	Retorna o caractere presente na posição do cursor.
<code>lcd_status ()</code>	Retorna o valor do status + contador de endereço
<code>imprime_string_lcd (s_caracteres)</code>	Envia uma <i>string</i> localizada na memória de programa para o <i>display LCD</i> . Sendo: <i>s_caracteres</i> – Ponteiro para a <i>string</i> .
<code>imprime_buffer_lcd (s_caracteres, tamanho_buffer)</code>	Envia uma matriz de dados para o <i>display LCD</i> . Sendo: <i>s_caracteres</i> – Ponteiro para o <i>buffer</i> . <i>tamanho_buffer</i> – Quantidade de dados que serão enviados.
<code>lcd_inicia (conf_1, conf_2, conf_3)</code>	Inicializa o <i>display LCD</i> alfanumérico. <i>conf_1</i> : valor localizado no campo <i>Display</i> da Tabela 8.2. <i>conf_2</i> : valor localizado na Tabela 8.3. <i>conf_3</i> : valor localizado no campo <i>Cursor</i> da Tabela 8.2

Para que o código funcione adequadamente em aplicações que utilizam pinos diferentes dos adotados por esta obra, é necessário modificar os pinos declarados na biblioteca. Outro ponto importante, é que a frequência de *clock* do oscilador deve ser informada, podendo ser diretamente modificada dentro desta, ou dentro do código do programa, inserindo o seguinte código:

```
#define Fosc frequência_do_oscilador_em_M
```



8.7 Projeto

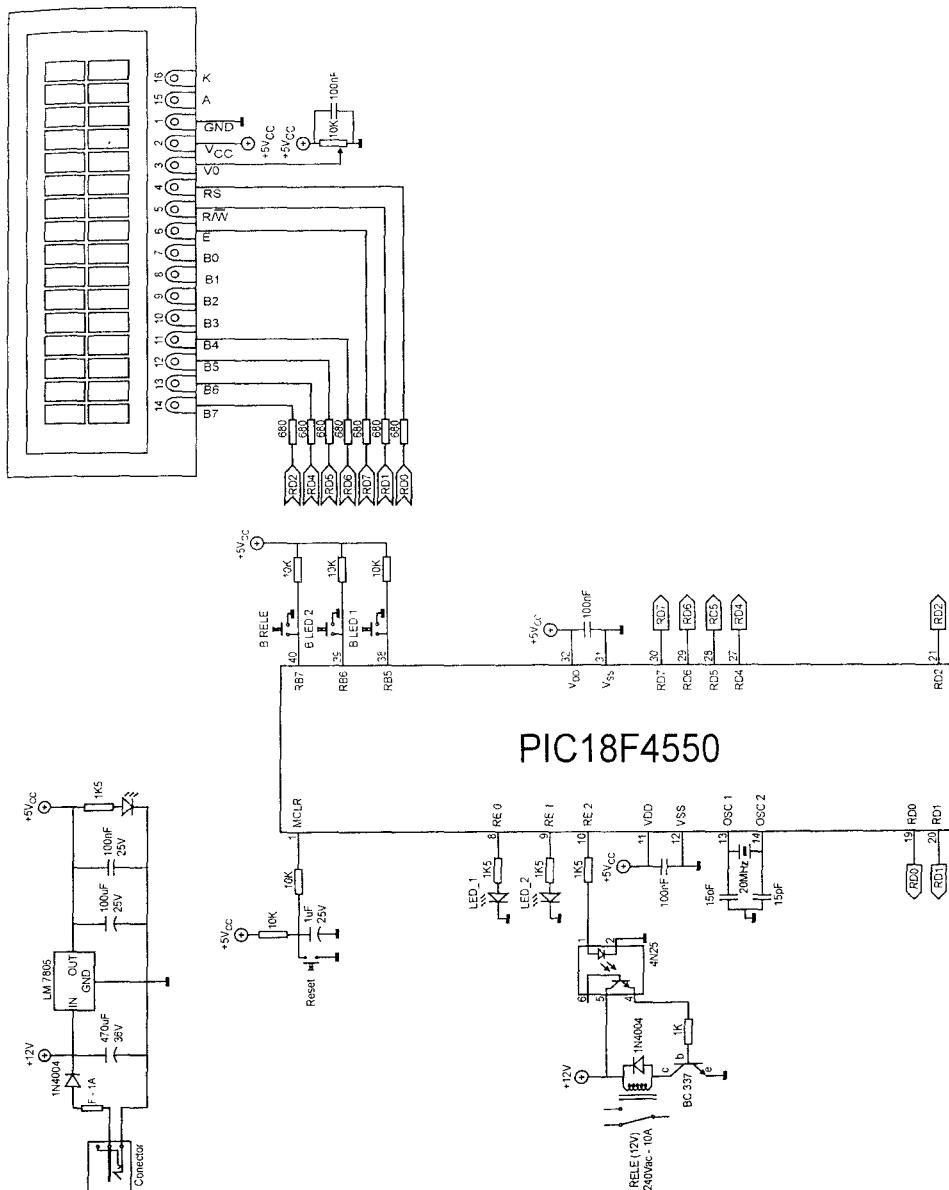


Figura 8.4: Circuito de controle do display LCD 2x16.

Código

```
/*
*****Projeto Capítulo 8 - Display LCD 2x16*****
**
** A rotina testa algumas funções de controle do LCD, tais como: escrita, L/D cursor e display, deslocamento de
** cursor e/ou mensagem etc
**
** Autor: Alberto Noboru Miyadaira
****/
```

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550
//#include <p18f455.h> //Código compatível com o PIC18F4455.
#include<delays.h> //Adiciona a biblioteca de delay
#include "C:\pic18\Biblioteca_Lcd_2x16.h" //Biblioteca contendo as funções do LCD
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)

#define Fosc 20 //Define a frequência do oscilador. Necessário para a biblioteca_lcd_2x16.h.

void gera_atraso_segundos(unsigned char atraso)
{
    do
    {
        Delay10KTCYx (20*Fosc/4); //Gera um delay de 0.2 segundos
        Delay10KTCYx (20*Fosc/4); //Gera um delay de 0.2 segundos
        Delay10KTCYx (20*Fosc/4); //Gera um delay de 0.2 segundos.
        Delay10KTCYx (20*Fosc/4); //Gera um delay de 0.2 segundos.
        Delay10KTCYx (20*Fosc/4); //Gera um delay de 0.2 segundos
        atraso--;
    }while (atraso>0);
}

void cria_caracteres_especiais(void)
{
    lcd_envia_controle (0, 0, 0x40, 2);
    //Caractere especial sugerido no livro.
    lcd_escreve_dado(0x01);
    lcd_escreve_dado(0x04);
    lcd_escreve_dado(0x1C);
    lcd_escreve_dado(0x1D),
    lcd_escreve_dado(0x1C);
    lcd_escreve_dado(0x04);
    lcd_escreve_dado(0x01);
    lcd_escreve_dado(0x00);
    //Outro caractere especial qualquer.
    lcd_escreve_dado(0x1F);
    lcd_escreve_dado(0x11);
    lcd_escreve_dado(0x11);
    lcd_escreve_dado(0x11);
    lcd_escreve_dado(0x11);
    lcd_escreve_dado(0x11);
    lcd_escreve_dado(0x1F);
    lcd_escreve_dado(0x00);
}

void main( ) //Função principal
```

```

{
unsigned char dado_lido;

TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída
TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída
TRISD = 0b00000000; //RD0 a RD7 – saída.
TRISE = 0b00000000; //RE0 a RE2 – saída.

PORTD = 0x00; //Coloca a porta D em 0V.
PORTE = 0x00; //Coloca a porta E em 0V.

lcd_inicia(0x28, 0x0f, 0x06); //Inicializa o display LCD alfanumérico com quatro linhas de dados.

cria_caracteres_especiais( );//Cria dois caracteres especiais.

while ( 1 )//Looping infinito
{
    lcd_posicao (1,1)// Desloca o cursor para a primeira coluna da primeira linha
    imprime_string_lcd("Microcontrolador"); //Imprime uma string no display
    lcd_posicao (1,3)// Desloca o cursor para a terceira coluna da primeira linha.
    dado_lido = lcd_le_dado (); //Lê o caractere presente na posição do cursor
    lcd_posicao (2,1)// Desloca o cursor para a primeira coluna da segunda linha.
    imprime_string_lcd("POS (1,3) - "); //Imprime uma string no display.
    lcd_escreve_dado(dado_lido); //Imprime o caractere correspondente à posição (1 3) 'c'

    gera_atraso_segundos (2),
    lcd_limpa_tela( ); // Limpa a tela do display e posiciona o cursor na linha um e coluna um.

    lcd_posicao (1,1);// Desloca o cursor para a primeira coluna da primeira linha
    imprime_string_lcd(" PIC18F4550 "); //Imprime uma string no display.

    lcd_posicao (2,7); // Desloca o cursor para a sétima coluna da segunda linha.
    lcd_escreve_dado('P'); //Imprime um caractere ASCII no display.
    lcd_escreve_dado('I'); //Imprime um caractere ASCII no display.
    lcd_escreve_dado('C'); //Imprime um caractere ASCII no display.

    gera_atraso_segundos (1);
    lcd_LD_cursor (0); // Desliga o cursor.
    gera_atraso_segundos (2);
    lcd_LD_cursor (1); // Desliga o display.
    gera_atraso_segundos (2);
    lcd_LD_cursor (2); // Liga o display com o cursor piscante.
    gera_atraso_segundos (2);

    lcd_desloca_cursor (1); // Desloca o cursor para a esquerda.
    gera_atraso_segundos (2);
    lcd_desloca_cursor (0); // Desloca o cursor para a direita.
    gera_atraso_segundos (2);

    lcd_desloca_mensagem (1); // Desloca a mensagem para a esquerda.
    gera_atraso_segundos (2);
    lcd_desloca_mensagem (0); // Desloca a mensagem para a direita.
    gera_atraso_segundos (2);

    lcd_limpa_tela( ); // Limpa a tela do display e posiciona o cursor na linha um e coluna um.
}

```

```
lcd_posicao (1,1); // Desloca o cursor para a primeira coluna da primeira linha  
imprime_string_lcd("C especial - "); //Imprime uma string no display.  
lcd_escreve_dado(0xC0); //Imprime o primeiro caractere especial. 0x00 é o endereço onde se encontra  
  
lcd_posicao (2,1); // Desloca o cursor para a primeira coluna da segunda linha  
imprime_string_lcd("C especial - "); //Imprime uma string no display.  
lcd_escreve_dado(0x01); //Imprime o primeiro caractere especial. 0x01 é o endereço onde se encontra.  
  
gera_atraso_segundos (2);  
lcd_limpa_tela(); // Limpa a tela do display e posiciona o cursor na linha um e coluna um  
  
imprime_string_lcd("Teste finalizado"); //Imprime uma string no display.  
gera_atraso_segundos (4);  
}
```

9

Interrupção

As interrupções desempenham um papel importante em sistemas microcontrolados, pois permitem suspender a qualquer momento a execução de uma linha de código e começar a executar uma rotina designada ao tratamento de uma determinada interrupção. Após finalizar a rotina, o processador retorna ao ponto do programa em que estava antes de a interrupção ser gerada.

Caracterizam-se pela alta velocidade de execução, uma vez que as interrupções são tratadas em hardware. As fontes de interrupções podem ser eventos externos, como a mudança de estado de um pino de entrada, como também ser decorrentes de eventos internos, como, por exemplo, conversão A/D, estouro de **TIMER**, estouro de **WDT** etc.

O diagrama representado na Figura 9.1 ilustra o comportamento de uma interrupção.

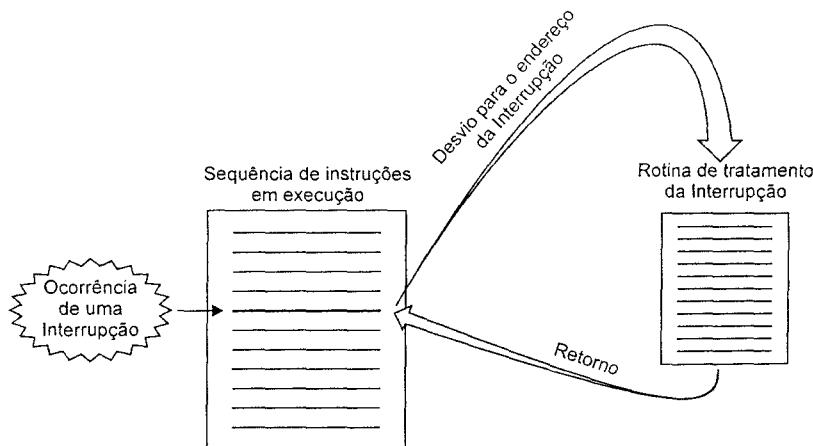


Figura 9.1: Interrupção.

O microcontrolador PIC18F4550 dispõe de múltiplas fontes de interrupção, sendo possível definir o nível de prioridade no atendimento de cada uma, selecionando-a como nível de prioridade alto (*high priority level*) ou nível de prioridade baixo (*low priority level*). Uma vez habilitada, se o dispositivo estiver atendendo uma interrupção de baixa prioridade, e em determinado momento ocorrer uma interrupção de alta prioridade, o controlador é desviado para o endereço dessa interrupção, e após tratá-la, volta a executar a rotina antecedente. Se a opção de prioridade de interrupção não estiver habilitada, elas são armazenadas no endereço 0008h. Veja a seguir o bloco lógico de interrupção do PIC18F4550.

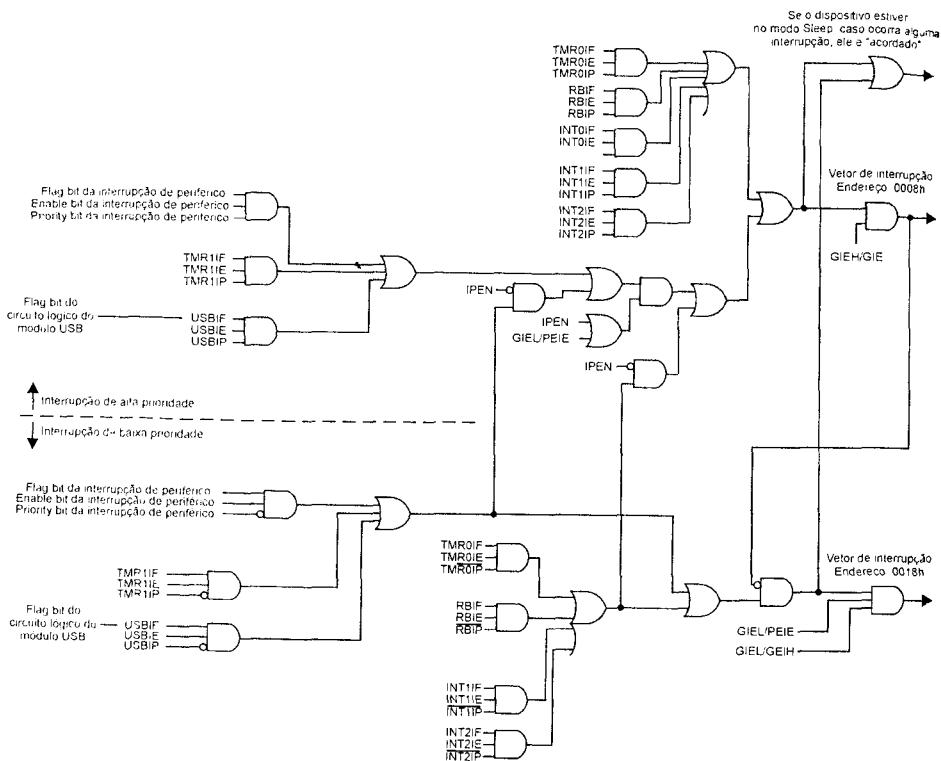


Figura 9.2: Bloco lógico de interrupção do PIC18F4550.

Diferente dos outros periféricos, o módulo USB é capaz de gerar uma ampla faixa de interrupções para muitos tipos de evento, e está equipado com sua própria lógica de interrupção, comentada no capítulo relacionado à USB.

9.1 Bits de Configuração da Interrupção

Vimos que o dispositivo pode ser configurado para tratar as interrupções sem levar em consideração o nível de prioridade (modo padrão), ou ser configurado para tratá-la de acordo com o nível de prioridade (alta ou baixa prioridade) selecionado para cada evento.

9.1.1 Interrupção com Nível de Prioridade

O primeiro passo para configurar o dispositivo para tratar as interrupções de acordo com o seu nível de prioridade é setar o bit IPEN (RCON<7> = 1) e habilitar as interrupções globalmente.

- **Bit GIEH (INTCON<7> = 1):** habilita as interrupções de alta prioridade.
- **Bit GIEL (INTCON<6> = 1):** habilita as interrupções de baixa prioridade.

Uma vez configurado, quando ocorrer uma interrupção, o programa é desviado para o endereço 000008h (High Priority) ou 000018h (Low Priority), de acordo com o nível de prioridade selecionado.



Quando uma interrupção é gerada, qualquer um dos dois bits GIEH ou GIEL é limpo.

9.1.2 Interrupção Sem Nível de Prioridade

O nível de prioridade no atendimento às interrupções é opcional. Por definição, o dispositivo está configurado para tratar as interrupções sem nível de prioridade. Para habilitar essa função manualmente, basta limpar o bit IPEN (RCON<7> = 0) e habilitar as interrupções globalmente por meio dos bits PEIE e GIE.

- **Bit GIE (INTCON<7> = 1):** habilita todas as fontes de interrupções.
- **Bit PEIE (INTCON<6> = 1):** habilita as fontes de interrupções de periféricos.

Note que os bits GIEH/GIE e GIEL/PEIE compartilham a mesma posição, ou seja, a função do bit 7 e do bit 6 do registro INTCON depende diretamente do modo como o dispositivo está configurado para tratar as interrupções (com/sem nível de prioridade).

Diferentemente do modo com nível de prioridade, todas as interrupções são desviadas para o endereço 000008h.



Quando uma interrupção é gerada, o bit GIE é limpo.

9.1.3 Bits de Configuração do Evento de Interrupção

Cada fonte de interrupção tem três bits associados, cujas funções são:

- **Bit de sinalização (Flag bit):** sinaliza a ocorrência da interrupção.
- **Bit de ativação (Enable bit):** ativa um determinado evento de interrupção.
- **Bit de prioridade (Priority bit):** define o nível de prioridade da interrupção.

Veja a seguir os registros de configuração para cada fonte de interrupção.

Tabela 9.1: Bits de configuração das fontes de interrupção.

Colisão de barramento		
Tipo	Registrador	Descrição
Enable bit	BCLIE (PIE2<3>)	1 - Habilita a interrupção de colisão no barramento. 0 - Desabilita a interrupção de colisão no barramento.
Flag bit	BCLIF (PIR2<3>)	1 - Ocorreu uma colisão de barramento. (Deve ser limpo por software) 0 - Nenhuma colisão de barramento ocorreu.
Priority bit	BCLIP (IPR2<3>)	1 - Alta prioridade. 0 - Baixa prioridade.
EUSART - Recepção		
Enable bit	RCIE (PIE1<5>)	1 - Habilita a interrupção de recepção da EUSART. 0 - Desabilita a interrupção de recepção da EUSART.
Flag bit	RCIF (PIR1<5>)	1 - Dado disponível no buffer de recepção (RCREG) da EUSART. 0 - Buffer de recepção (RCREG) da EUSART está vazio.
Priority bit	RCIP (IPR1<5>)	1 - Alta prioridade. 0 - Baixa prioridade.

EUSART - Transmissão

<i>Enable bit</i>	TXIE (PIE1<4>)	1 - Habilita a interrupção de transmissão da EUSART. 0 - Desabilita a interrupção de transmissão da EUSART.
<i>Flag bit</i>	TXIF (PIR1<4>)	1 - Buffer de transmissão (TXREG) da EUSART está vazio 0 - Buffer de transmissão (TXREG) da EUSART está cheio
<i>Priority bit</i>	TXIP (IPR1<4>)	1 - Alta prioridade. 0 - Baixa prioridade.

Falha do oscilador

<i>Enable bit</i>	OSCFIE (PIE2<7>)	1 - Habilita a interrupção de falha no oscilador. 0 - Desabilita a interrupção de falha no oscilador.
<i>Flag bit</i>	OSCFIF (PIR2<7>)	1 - Oscilador do sistema falhou, a entrada de <i>clock</i> mudou para INTOSC. (Deve ser limpo por <i>software</i>) 0 - Oscilador do sistema operando normalmente
<i>Priority bit</i>	OSCFIP (IPR2<7>)	1 - Alta prioridade. 0 - Baixa prioridade.

Interrupção externa - INT0

<i>Enable bit</i>	INT0IE (INTCON<4>)	1 - Habilita interrupção externa INT0. 0 - Desabilita interrupção externa INT0.
<i>Flag bit</i>	INT0IF (INTCON<1>)	1 - Ocorreu interrupção externa INT0. (Deve ser limpo por <i>software</i>) 0 - Não ocorreu interrupção externa INT0.
<i>Priority bit</i>	'-	Não suporta nível de prioridade.
<i>Edge</i>	INTEDG0 (INTCON2<6>)	1 - Interrupção na borda de subida. 0 - Interrupção na borda de descida.

Interrupção externa - INT1

Tipo	Registrador	Descrição
<i>Enable bit</i>	INT1IE (INTCON3<3>)	1 - Habilita interrupção externa INT1. 0 - Desabilita interrupção externa INT1.
<i>Flag bit</i>	INT1IF (INTCON3<0>)	1 - Ocorreu interrupção externa INT1. (Deve ser limpo por <i>software</i>) 0 - Não ocorreu interrupção externa INT1.
<i>Priority bit</i>	INT1IP (INTCON3<6>)	1 - Alta prioridade. 0 - Baixa prioridade.
<i>Edge</i>	INTEDG1 (INTCON2<5>)	1 - Interrupção na borda de subida. 0 - Interrupção na borda de descida.

Interrupção externa - INT2

<i>Enable bit</i>	INT2IE (INTCON3<4>)	1 - Habilita interrupção externa INT2. 0 - Desabilita interrupção externa INT2.
<i>Flag bit</i>	INT2IF (INTCON3<1>)	1 - Ocorreu interrupção externa INT2. (Deve ser limpo por <i>software</i>) 0 - Não ocorreu interrupção externa INT2.
<i>Priority bit</i>	INT2IP (INTCON3<7>)	1 - Alta prioridade. 0 - Baixa prioridade.
<i>Edge</i>	INTEDG2 (INTCON2<4>)	1 - Interrupção na borda de subida. 0 - Interrupção na borda de descida.

Master Synchronous Serial Port (MSSP) - SPI e I ² C		
<i>Enable bit</i>	SSPIE (PIE1<3>)	1 - Habilita a interrupção do MSSP. 0 - Desabilita a interrupção do MSSP.
<i>Flag bit</i>	SSPIF (PIR1<3>)	1 - Transmissão/Recepção completada. (Deve ser limpo por software) 0 - Esperando para transmitir/receber.
<i>Priority bit</i>	SSPIP (IPR1<3>)	1 - Alta prioridade. 0 - Baixa prioridade.
Memória EEPROM/FLASH		
<i>Enable bit</i>	EEIE (PIE2<4>)	1 - Habilita a interrupção de escrita na EEPROM/FLASH. 0 - Desabilita a interrupção de escrita na EEPROM/FLASH.
<i>Flag bit</i>	EEIF (PIR2<4>)	1 - Operação de escrita finalizada. (Deve ser limpo por software) 0 - Operação de escrita em andamento ou não foi iniciada
<i>Priority bit</i>	EEIP (IPR2<4>)	1 - Alta prioridade. 0 - Baixa prioridade.
Módulo CCP1		
<i>Enable bit</i>	CCP1IE (PIE1<2>)	1 - Habilita a interrupção do módulo CCP1. 0 - Desabilita a interrupção do módulo CCP1.
<i>Flag bit</i>	CCP1IF (PIR1<2>)	Capture: 1 - Ocorreu captura do registro TMR1 ou TMR3. (Deve ser limpo por software) 0 - Não ocorreu captura do registro TMR1 ou TMR3. Compare: 1 - Ocorreu combinação do registro TMR1 ou TMR3. (Deve ser limpo por software) 0 - Não ocorreu combinação do registro TMR1 ou TMR3. PWM: Não é usado neste modo. •
<i>Priority bit</i>	CCP1IP (IPR1<2>)	1 - Alta prioridade. 0 - Baixa prioridade.
Módulo CCP2		
Tipo	Registrador	Descrição
<i>Enable bit</i>	CCP2IE (PIE2<0>)	1 - Habilita a interrupção do módulo CCP2. 0 - Desabilita a interrupção do módulo CCP2.
<i>Flag bit</i>	CCP2IF (PIR2<0>)	Capture: 1 - Ocorreu captura do registro TMR1 ou TMR3. (Deve ser limpo por software) 0 - Não ocorreu captura do registro TMR1 ou TMR3. Compare: 1 - Ocorreu combinação do registro TMR1 ou TMR3. (Deve ser limpo por software) 0 - Não ocorreu combinação do registro TMR1 ou TMR3. PWM: Não é usado neste modo.
<i>Priority bit</i>	CCP2IP (IPR2<0>)	1 - Alta prioridade. 0 - Baixa prioridade.
Módulo comparador		
<i>Enable bit</i>	CMIE (PIE2<6>)	1 - Habilita a interrupção do comparador. 0 - Desabilita a interrupção do comparador.
<i>Flag bit</i>	CMIF (PIR2<6>)	1 - Entrada do comparador mudou. (Deve ser limpo por software) 0 - Entrada do comparador não mudou.
<i>Priority bit</i>	CMIP (IPR2<6>)	1 - Alta prioridade. 0 - Baixa prioridade.

Módulo conversor A/D		
<i>Enable bit</i>	ADIE (PIE1<6>)	1 - Habilita a interrupção do conversor A/D. 0 - Desabilita a interrupção do conversor A/D.
<i>Flag bit</i>	ADIF (PIR1<6>)	1 - Conversão A/D completada. (Deve ser limpo por software) 0 - Conversão A/D não completada.
<i>Priority bit</i>	ADIP (IPR1<6>)	1 - Alta prioridade. 0 - Baixa prioridade.
Módulo de detecção de alta/baixa voltagem		
<i>Enable bit</i>	HLVDIE (PIE2<2>)	1 - Habilita a interrupção de detecção de alta/baixa voltagem. 0 - Desabilita a interrupção de detecção de alta/baixa voltagem.
<i>Flag bit</i>	HLVDIF (PIR2<2>)	1 - Ocorreu uma condição de alta/baixa voltagem. (Deve ser limpo por software) 0 - Não ocorreu uma condição de alta/baixa voltagem.
<i>Priority bit</i>	HLVDIP (IPR2<2>)	1 - Alta prioridade. 0 - Baixa prioridade.
Módulo USB		
<i>Enable bit</i>	USBIE (PIE2<5>)	1 - Habilita a interrupção da USB. 0 - Desabilita a interrupção da USB.
<i>Flag bit</i>	USBFIF (PIR2<5>)	1 - USB solicitou uma interrupção. (Deve ser limpo por software) 0 - Sem solicitação de interrupção pela USB.
<i>Priority bit</i>	USBIP (IPR2<5>)	1 - Alta prioridade. 0 - Baixa prioridade.
RB		
Tipo	Registrador	Descrição
<i>Enable bit</i>	RBIE (INTCON<3>)	1 - Habilita interrupção por mudança de estado nos pinos RB4:RB7. 0 - Desabilita interrupção por mudança de estado nos pinos RB4:RB7.
<i>Flag bit</i>	RBFIF (INTCON<0>)	1 - Pelo menos um dos pinos RB4:RB7 mudou de estado. (Deve ser limpo por software) 0 - Nenhum dos pinos RB4:RB7 mudou de estado.
<i>Priority bit</i>	RBIP (INTCON2<0>)	1 - Alta prioridade. 0 - Baixa prioridade.
Streaming Parallel Port (SPP)		
<i>Enable bit</i>	SPIE (PIE1<7>)	1 - Habilita a interrupção de escrita/leitura do SPP. 0 - Desabilita a interrupção de escrita/leitura do SPP.
<i>Flag bit</i>	SPIIF (PIR1<7>)	1 - Uma operação de escrita ou leitura foi executada. (Deve ser limpo por software) 0 - Não ocorreu nenhuma operação de escrita ou leitura.
<i>Priority bit</i>	SPIIP (IPR1<7>)	1 - Alta prioridade. 0 - Baixa prioridade.
TIMER 0		
<i>Enable bit</i>	TMR0IE (INTCON<5>)	1 - Habilita interrupção por estouro do contador do TIMER 0. 0 - Desabilita interrupção por estouro do contador do TIMER 0.
<i>Flag bit</i>	TMR0IF (INTCON<2>)	1 - Registro TMR0 estourou. (Deve ser limpo por software) 0 - Registro TMR0 não estourou.
<i>Priority bit</i>	TMR0IP (INTCON2<2>)	1 - Alta prioridade. 0 - Baixa prioridade.

TIMER 1		
<i>Enable bit</i>	TMR1IE (PIE1<0>)	1 - Habilita a interrupção de estouro do TMR1. 0 - Desabilita a interrupção de estouro do TMR1.
<i>Flag bit</i>	TMR1IF (PIR1<0>)	1 - Registro TMR1 estourou. (Deve ser limpo por software) 0 - Registro TMR1 não estourou.
<i>Priority bit</i>	TMR1IP (IPR1<0>)	1 - Alta prioridade. 0 - Baixa prioridade
TIMER 2		
<i>Enable bit</i>	TMR2IE (PIE1<1>)	1 - Habilita a interrupção de combinação do TMR2 e PR2. 0 - Desabilita a interrupção de combinação do TMR2 e PR2.
<i>Flag bit</i>	TMR2IF (PIR1<1>)	1 - O valor do registro TMR2 é igual ao valor do registro PR2. (Deve ser limpo por software) 0 - O valor do registro TMR2 é diferente do valor do registro PR2.
<i>Priority bit</i>	TMR2IP (IPR1<1>)	1 - Alta prioridade 0 - Baixa prioridade.
TIMER 3		
<i>Enable bit</i>	TMR3IE (PIE2<1>)	1 - Habilita a interrupção de estouro do TMR3. 0 - Desabilita a interrupção de estouro do TMR3.
<i>Flag bit</i>	TMR3IF (PIR2<1>)	1 - Registro TMR3 estourou. (Deve ser limpo por software) 0 - Registro TMR3 não estourou.
<i>Priority bit</i>	TMR3IP (IPR2<1>)	1 - Alta prioridade 0 - Baixa prioridade

O acesso aos registros é feito de acordo com a sintaxe a seguir.

Sintaxe

<nome_registro>bits.<nome_bit>

Sendo:

- **nome_registro**: nome do registro associado ao evento de interrupção.
- **nome_bit**: nome do *bit* associado ao registro.

Este formato pode ser usado tanto para ler ou escrever dados nos registros.

Exemplo

RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo.

INTCONbits.GIEH = 1; //Habilita todas as interrupções de alta prioridade.

INTCONbits.GIEL = 1; //Habilita todas as interrupções de baixa prioridade.

INTCONbits.INT0IE = 1; //Ativa a interrupção externa INT0 (RB0).

INTCON2bits.INTEDG0 = 0; //Interrupção externa INT0 na borda de descida.

INTCONbits.INT0IF = 0; // Limpa o flag bit da interrupção externa INT0.

9.2 Comportamento da Interrupção

Uma interrupção suspende a execução de uma aplicação, salva as informações de contexto e desvia o controle para uma rotina de serviço de interrupção (ISR) para que o evento possa ser processado. Ao finalizar a execução da ISR, as informações de contexto anteriormente salvas são restauradas e a execução normal da aplicação recomeça. Contextos são recursos da CPU que o programa usa em um determinado momento, podendo ser registros para memória ou periféricos. O contexto mínimo salvo e restaurado por uma interrupção consiste nos registros **WREG**, **Status** e **BSR**.

As interrupções podem ser tratadas de dois modos distintos. O primeiro modo não leva em consideração o nível de prioridade (modo-padrão) e no segundo, o nível de prioridade da interrupção é avaliado.

Se o dispositivo estiver configurado para atender às interrupções, como modo-padrão (*IPEN* = 0), a ocorrência de qualquer interrupção (*Flag bit* = 1) desvia a execução do programa para o endereço do vetor de interrupção 000008h; caso contrário, o programa é desviado para o endereço do vetor de interrupção 000008h (*Priority bit* = 1) ou 000018h (*Priority bit* = 0), de acordo com o *Priority bit* selecionado pela fonte.

Em ambos os casos, quando uma interrupção é gerada, o endereço de retorno é inserido na pilha, o *Program Counter* (PC) é carregado com o endereço do vetor de interrupção (000008h ou 000018h) e os registros **WREG**, **Status** e **BSR** são salvos na pilha de retorno rápido.

A pilha de retorno rápido possui apenas um nível e é composta pelos registros **WREG**, **Status** e **BSR**. É normalmente utilizada para prover "retorno rápido" de interrupção e não pode ser lida nem escrita pelo usuário. Sempre que ocorre uma interrupção, o valor correto dos registros é carregado para dentro da pilha, sendo retirado toda vez que a instrução **RETFIE 0x01** é executada.

Devido ao fato de que a pilha de retorno rápido possui somente um nível e toda chamada de interrupção carrega o valor correto dos registros nessa pilha, se a prioridade de interrupção estiver desabilitada, a instrução **RETFIE 0x01** pode ser usada para retornar de qualquer interrupção. No entanto, se a prioridade de interrupção estiver habilitada, os registros salvos na pilha de retorno rápido não podem ser usados com segurança para as interrupções de baixa prioridade, pois eles podem ser sobreescritos caso ocorra uma interrupção de alta prioridade durante a sua execução. Neste caso, o usuário deve salvar os registros via *software* durante a interrupção de baixo nível e carregá-los por *software* antes de retornar da interrupção pela instrução **RETFIE 0x00**, que não retorna os registros armazenados na pilha de retorno rápido, mas simplesmente reabilita as interrupções.

Prioridade de interrupção habilitada (*IPEN* = 1)

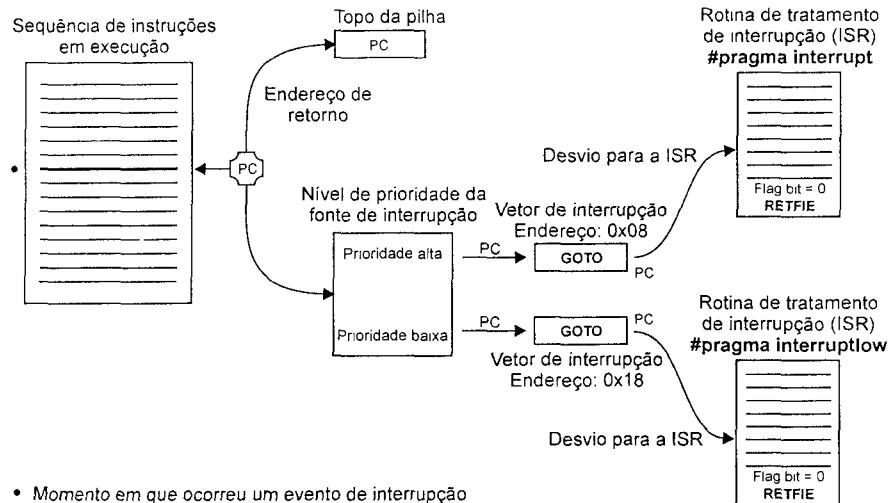
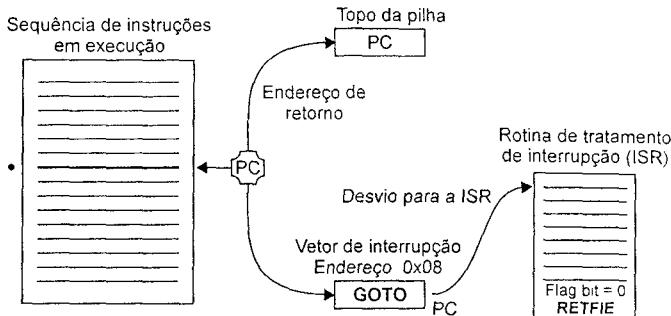


Figura 9.3: Comportamento da interrupção com prioridade de interrupção.

Prioridade de interrupção desabilitada (IPEN = 0)



- Momento em que ocorreu um evento de interrupção

Figura 9.4: Comportamento da interrupção sem prioridade de interrupção.

Quando o PC é desviado para o endereço do vetor de interrupção, normalmente é executada uma instrução **GOTO** com a finalidade de desviar o programa para uma rotina de serviço de interrupção (ISR), que pode ser utilizada para localizar a fonte de interrupção, por meio do *Flag bit*, e então executar uma rotina destinada a um tipo específico de interrupção.

Uma ISR é como qualquer outra função C. A principal diferença é que ela não tem parâmetros de entrada nem retorna valores, e as variáveis globais acessadas pela ISR e função principal devem ser declaradas como **volatile**. Elas devem ser executadas somente por interrupções de *hardware* e nunca por funções C. Após executar as instruções de tratamento da interrupção, antes de sair da ISR é necessário limpar o *Flag bit* da fonte de interrupção e executar a função de retorno de interrupção **RETFIE**, em vez do **RETURN**, pois a instrução **RETFIE** reabilita as interrupções (GIE/GIEH=1 ou PEIE/GIEL=1).



O *Flag bit* deve ser limpo por *software* antes de reabilitar as interrupções, a fim de evitar que a interrupção seja gerada antes de sair da rotina de tratamento de interrupção.

Não use a instrução **MOVFF** para modificar registros de controle de interrupção, enquanto qualquer interrupção estiver sendo atendida, pois pode causar problemas no comportamento do dispositivo.

9.3 Diretiva de Interrupção

O MPLAB® C18 comporta duas diretivas **#pragma** para o tratamento das interrupções. A diretiva **#pragma interrupt** declara uma função para ser uma rotina de serviço de interrupção (ISR) de alta prioridade, enquanto **#pragma interruptlow** declara uma função para ser uma ISR de baixa prioridade.

Sintaxe

```
#pragma interrupt nome_função nome_seção_tmp save = lista_s nosave = lista_ns
```

```
#pragma interruptlow nome_função nome_seção_tmp save = lista_s nosave = lista_ns
```

Sendo:

- **nome_função:** nome da função servindo como ISR.
- **nome_seção_tmp:** nome da seção de dado temporário do ISR. Se não for informado o nome da seção, as variáveis temporárias são criadas em uma seção **udata** de nome **fname_tmp**.

- **lista_s:** nomes de variáveis e/ou seções de dados, separados por vírgula ',', que serão salvos e recuperados pela função.
- **lista_ns:** nomes de variáveis e/ou seções de dados, separados por vírgula ',', que não serão salvos nem recuperados pela função.

Por definição, quando uma ISR é chamada, o compilador MPLAB® C18 preserva os recursos gerenciados por ele. Veja a tabela a seguir.

Tabela 9.2: Recursos preservados pelo compilador.

Recursos gerenciados pelo compilador	Uso primário
PC	Controle de execução.
WREG	Cálculos intermediários
STATUS	Resultado dos cálculos.
BSR	Seleção de banco.
PROD	Resultado de multiplicação, valores de retorno e cálculos intermediários.
Seção .tmpdata	Cálculos intermediários.
FSR0	Ponteiros para a RAM.
FSR1	Ponteiro de pilha.
FSR2	Ponteiro de estrutura.
TBLPTR	Acesso de valores na memória de programa.
TABLAT	Acesso de valores na memória de programa.
PCLATH	Chamada de ponteiro de função.
PCLATU	Chamada de ponteiro de função.
Seção MATH_DATA	Argumentos, valores de retorno e localização temporária para as funções da biblioteca <i>math</i> .

Além dos recursos preservados pelo compilador, também é possível adicionar símbolos para serem salvos e recuperados pela função, através da cláusula **save=**. No exemplo a seguir, *isr_alto* é o nome da função contendo a ISR, *var_global* é uma variável global que será salva e recuperada e *meu_dado* é o nome da seção definida pelo usuário, que também será salva e recuperada.

```
#pragma interrupt isr_alto save = var_global, section("meu_dado")
```

A cláusula **nosave=** permite que os recursos gerenciados pelo compilador sejam especificados para serem usados somente dentro da ISR. O exemplo a seguir especifica que os registros **FSR0** e **PROD** não precisam ser salvos nem recuperados pela função de interrupção de alta prioridade (*isr_alto*).

```
#pragma interrupt isr_alto nosave = FSR0, PROD
```

Exemplo

```
void ISR_alta_prioridade(void); //Protótipo da função.
void ISR_baixa_prioridade(void); //Protótipo da função.
...
#pragma code int_alta = 0x08 //Vetor de interrupção de alta prioridade.
```

```
void int_alta (void)
{
    _asm
        GOTO ISR_alta_prioridade //Desvia o programa para a função ISR_alta_prioridade.
    _endasm
}
#pragma code

#pragma interrupt ISR_alta_prioridade
void ISR_alta_prioridade (void)
{
    //Rotina de tratamento do evento de interrupção.
}

#pragma code int_baixa = 0x18 //Vetor de interrupção de baixa prioridade.
void int_alta (void)
{
    _asm
        GOTO ISR_baixa_prioridade //Desvia o programa para a função ISR_baixa_prioridade.
    _endasm
}
#pragma code

#pragma interrupt ISR_baixa_prioridade
void ISR_baixa_prioridade (void)
{
    //Rotina de tratamento do evento de interrupção
}
```

9.4 Período de Latência

O período de latência de uma interrupção é o tempo entre a ocorrência de uma interrupção e a execução da primeira instrução da ISR. Existem basicamente três fontes associadas a esse período:

- Tempo usado pela CPU para reconhecer a interrupção e desviar para o primeiro endereço do vetor de interrupção.
- Tempo usado para executar o código no vetor de interrupção que vai desviar o controle para a ISR.
- Tempo usado pelo compilador MPLAB® C18 para salvar os recursos gerenciados por ele e os dados inseridos na cláusula **save=**.

9.5 Projeto

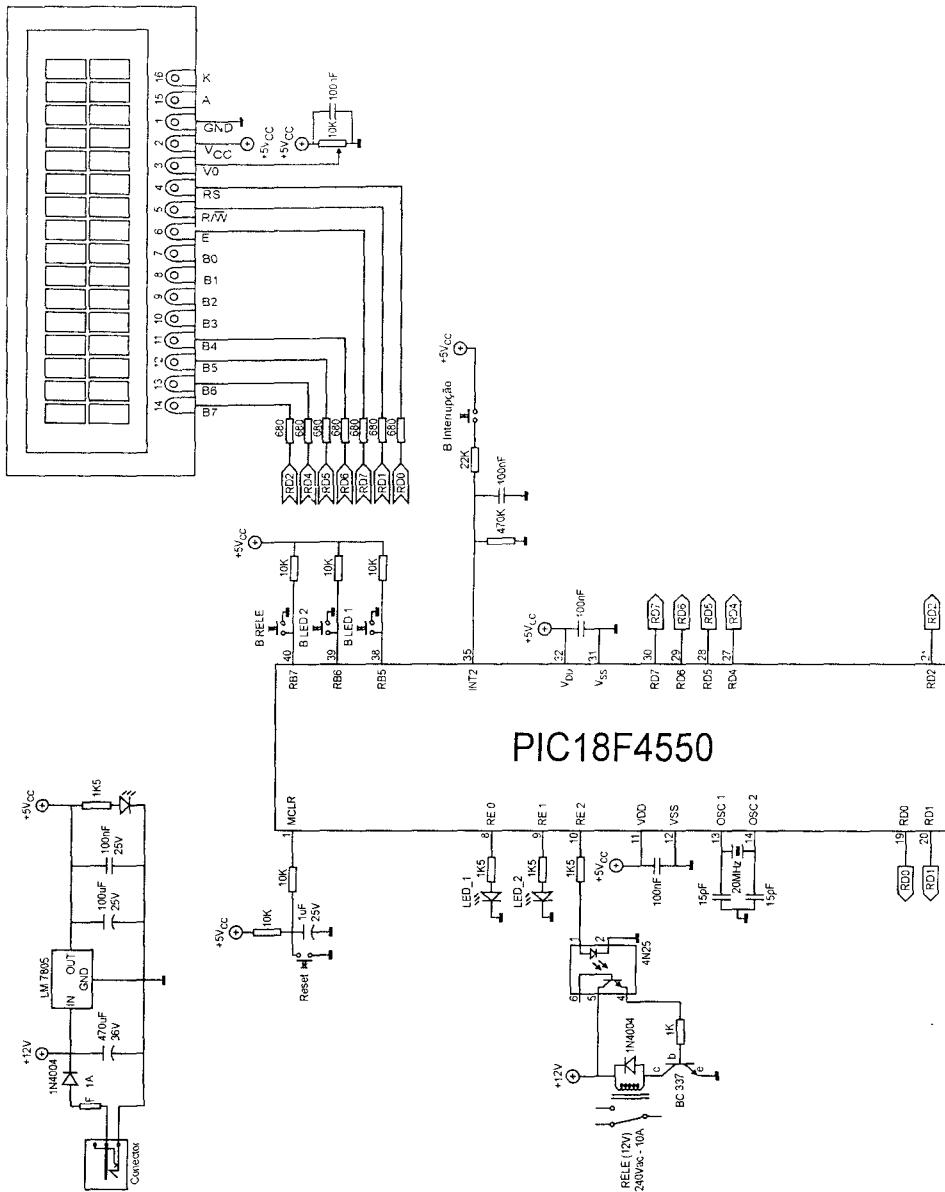


Figura 9.5: Circuito de interrupção externa.

Código

```
*****Projeto Capítulo 9 – INTERRUPÇÕES*****
**
** Este programa permite que a interrupção externa seja acessada somente quatro vezes.
** Após a quarta vez, a interrupção é desativada.
** Sempre que a interrupção é acessada o status do relé e do LED é invertido
**
** Autor: Alberto Noboru Miyadaira
*****
```

#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550
 #include <adc.h> //Adiciona a biblioteca de funções para o módulo conversor A/D.
 #include <delays.h> //Adiciona a biblioteca de delay.
 #include <stdio.h> //Biblioteca padrão de entrada e saída.
 #include "C:\pic18\oteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.
 #include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)

void ISR_alta_prioridade(void) //Protótipo da função de interrupção.

unsigned char s_led=0, s_rele=0; //Declara duas variáveis globais do tipo unsigned char.
 unsigned char contador=0; //Declara uma variável global do tipo unsigned char.
 unsigned char buffer[16]; //Declara um buffer de 16 posições do tipo unsigned char.

#define LED LATExbits.LATE0 //Define outro nome para a estrutura.
#define RELE LATExbits.LATE2 //Define outro nome para a estrutura.

#define Fosc 20 //Define a frequência do oscilador

#pragma code int_alta=0x08 //Vetor de interrupção de alta prioridade. ou padrão.
void int_alta (void)
{
 _asm GOTO ISR_alta_prioridade _endasm //Desvia o programa para a função ISR_alta_prioridade
}
#pragma code

#pragma interrupt ISR_alta_prioridade
void ISR_alta_prioridade(void)
{
 if (contador < 4) //Verifica se o valor do contador é menor que 4.
 {
 RELE = ~RELE; //Inverte o estado do relé.
 LED = ~LED; //Inverte o estado do LED.
 INTCON3bits.INT2IF = 0; //Limpa o flag bit da interrupção externa INT2.
 contador++; //Incrementa o contador em 1, sempre que a interrupção for gerada.
 }
 else
 {
 INTCON3bits.INT2IF = 0; //Limpa o flag bit da interrupção externa INT2.
 _asm RETURN 0x01 _endasm //Sai da interrupção e não reabilita a interrupção.
 }
}

void gera_atraso_segundos(unsigned char atraso)
{
 do
 {
 Delay10KTCYx (20*Fosc/4); //Gera um delay de 0,2 segundo.
 Delay10KTCYx (20*Fosc/4); //Gera um delay de 0,2 segundo.
}

```

Delay10KTCYx (20*fosc/4); //Gera um delay de 0,2 segundo.
Delay10KTCYx (20*fosc/4); //Gera um delay de 0,2 segundo.
Delay10KTCYx (20*fosc/4); //Gera um delay de 0,2 segundo.
atraso--;
}while (atraso>0);
}

void main( ) //Função principal
{
    TRISA = 0b00011111; // RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11100111; // RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111; // RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0b00000000; // RD0 a RD7 – saída.
    TRISE = 0b00000000; // RE0 a RE2 – saída.

//Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital (Capítulo 13)
//Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
OpenADC (0x00, 0x00, ADC_0ANA), //Requer a biblioteca adc.h.
CloseADC (); //Requer a biblioteca adc.h.

PORTD = 0x00; //Coloca a porta D em 0V.
PORTE = 0x00; //Coloca a porta E em 0V.

lcd_inicia(0x28, 0XF, 0x06); // Inicializa o display LCD alfanumérico com quatro linhas de dados.
lcd_LD_cursor (0)// Desliga o cursor.
lcd_posicao (1,3); // Desloca o ponteiro para a coluna três da primeira linha.
imprime_string_lcd("Interrupcao:"); // Imprime uma string no display.
lcd_posicao (2,1); // Desloca o ponteiro para a coluna um da segunda linha.
imprime_string_lcd(" EXT-Pino INT2 "); // Imprime uma string no display.
gera_atraso_segundos (2); // Gera um atraso de dois segundos.

INTCON2bits.INTEDG2 = 0; // Interrupção externa INT2 na borda de descida.
INTCON3bits.INT2IF = 0; // Limpa o flag bit da interrupção externa INT2.
INTCON3bits.INT2IP = 1; // Alta prioridade.
INTCON3bits.INT2IE = 1; // Ativa a interrupção externa INT2 (RB2).

RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo.
INTCONbits.GIEH = 1;//Habilita todas as interrupções de alta prioridade.
INTCONbits.GIEL = 0;//Desabilita todas as interrupções de baixa prioridade.

while ( 1 )//Looping infinito.
{
    lcd_posicao (1,1); // Desloca o cursor para a L=1 e C=1.
    imprime_string_lcd(" EXT-Pino INT2 "); // Imprime uma string no display.
    lcd_posicao (2,1); // Desloca o cursor para a L=2 e C=1.
    sprintf(buffer, " Contador = %03u ",contador); // Coloca a string formatada dentro do buffer.
    imprime_buffer_lcd(buffer,16); // Imprime o conteúdo do buffer no display LCD.
    Delay10KTCYx (100); // Gera um delay de 200 milissegundos. 100*10.000*0,2us= 200ms
}
}

```

10

USART

A USART (*Universal Synchronous Asynchronous Receiver Transmitter*), também conhecida como interface de comunicação serial, pode ser configurada para trabalhar no modo assíncrono ou síncrono. O modo assíncrono (*full-duplex*) é normalmente utilizado quando se deseja comunicar com computadores pessoais. e o modo síncrono (*half-duplex*) permite a comunicação com conversores A/D, EEPROM serial etc.

A USART é um circuito eletrônico responsável pela interface entre o dispositivo e a porta serial. Na transmissão, o dispositivo envia os dados para a USART de forma paralela e a USART se encarrega de transmiti-los *bit a bit* pela comunicação serial. Na recepção, a USART realiza a operação contrária, ou seja, recebe a mensagem *bit a bit* e a entrega de forma paralela para o dispositivo.

Uma aplicação muito comum desse módulo é a comunicação assíncrona RS-232. Em seguida, acompanhe uma breve introdução sobre as características e o modo de funcionamento desse protocolo.

10.1 Protocolo RS-232

O protocolo RS-232 é um padrão de comunicação serial criado pela EIA (*Electronic Industries Association*) para a comunicação entre um DTE (terminal de dados) e um DCE (um comunicador de dados), também conhecida como EIA-232. Normalmente, o pacote enviado é constituído de 10 ou 11 bits, dos quais 8 bits constituem a mensagem, 1 bit de início (*Start Bit*), 1/1.5/2 bits de parada (*Stop Bit*) e 1 bit de paridade (*Parity Bit*) para o controle de erro.

As portas seriais normalmente utilizam conectores do tipo DB9, DB25 e RJ45. O conector DB9 é o mais utilizado na atualidade e a Figura 10.1 ilustra-o com suas respectivas pinagens.

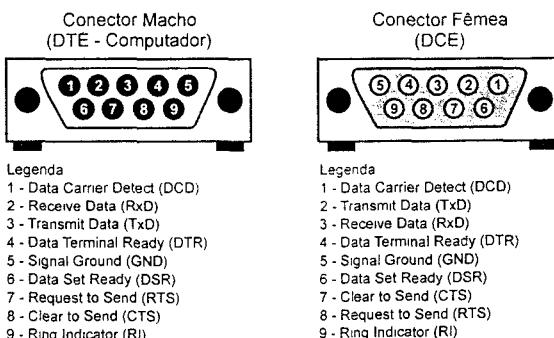


Figura 10.1: Conector DB9.

10.1.1 Funcionamento do Protocolo RS-232

Quando o canal não está transmitindo nenhum dado, ele apresenta um nível lógico '1'. Para sinalizar o início de uma comunicação, o transmissor coloca o canal em nível lógico '0' (*Start bit*), informando ao receptor que uma transmissão foi iniciada. Imediatamente o receptor dá início à contagem de *clock*, internamente, para receber os dados.

Após o envio do *start bit*, o transmissor manda a mensagem composta por 8bits de acordo com a tabela ASCII (Apêndice A), iniciando pelo *bit* menos significativo (Lsb), seguido de um *bit* de paridade (opcional) e de um *Stop bit*, sinalizando ao receptor o fim da transmissão. As taxas de transmissão comumente utilizadas são 300bps, 600bps, 1200bps, 2400bps, 4800bps, 9600bps, 19200bps etc.

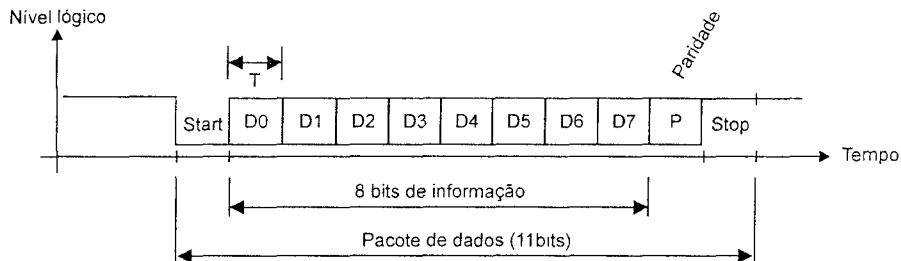


Figura 10.2: Transmissão RS-232 assíncrona.

O *bit* de paridade P (opcional) é responsável pelo controle de erro. Ele assume o valor '0' se a mensagem possuir um número par de *bits* iguais a 1; caso contrário, ele será igual a '1'. Para melhor comprehendê-lo, veja os exemplos a seguir.

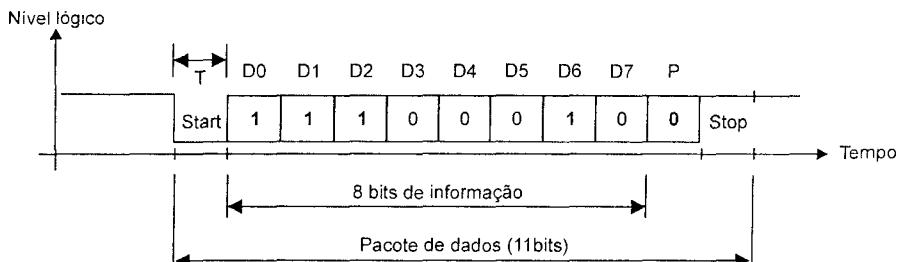


Figura 10.3: Bit de Paridade (P) igual a '0'.

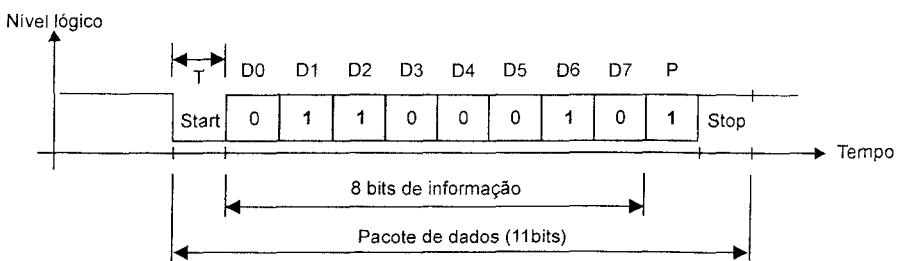


Figura 10.4: Bit de Paridade (P) igual a '1'.

O período de cada *bit* é definido por T, tempo que pode ser calculado, uma vez que a taxa de transmissão já esteja definida. Suponhamos que a taxa de transmissão seja 9600bps, logo $T = 1/9600 = 104,17\mu s$

A comunicação RS-232 é assíncrona. Isso significa que para efetuar a comunicação de dados entre os dispositivos, o transmissor e o receptor devem estar configurados a uma mesma taxa de transmissão/recepção, ou seja, se a transmissão está sendo efetuada a uma taxa de 9600bps, então o receptor deve estar configurado para receber os dados a 9600bps. Logo, se as velocidades de transmissão e recepção não forem compatíveis, os dados recebidos não serão interpretados corretamente.

10.1.2 Níveis Lógicos da Interface RS-232

Na interface RS-232 os níveis lógicos são representados da seguinte maneira:

- 0: tensão entre +3V a +25V
- 1: tensão entre -3V a -25V
- Indefinido (região de transição): tensão entre -3V a +3V

Note que os níveis de tensão utilizados na comunicação RS-232 não são compatíveis com os níveis CMOS/TTL. Para conectar o microcontrolador a uma porta serial do computador, é necessário utilizar conversores de nível, capazes de converter o nível TTL/CMOS em RS-232 e vice-versa. Entre os mais conhecidos estão o MAX232 e o MAX3232.

10.2 Módulo EUSART do PIC18F4550

O módulo USART do microcontrolador PIC18 pode ser utilizado no modo assíncrono ou síncrono. Quando configurado como assíncrono, ele utiliza os pinos RX (recepção) e TX (transmissão) para comunicação, enquanto no modo síncrono são utilizados os pinos CK (clock) e DT (dado). Ele pode ser configurado para receber/transmitir 8bits ou 9bits.

Vemos na sequência as funções adicionais desse módulo e como configurá-las.

10.2.1 Funções Adicionais da EUSART

Alguns microcontroladores da família PIC18, incluindo o PIC18F4550, possuem o módulo EUSART (*Enhanced Universal Synchronous Asynchronous Receiver Transmitter*) que é uma USART com algumas funções adicionais, tais como:

Modo Assíncrono (*full-duplex*)

- Auto-wake-up na recepção de caractere.
- Calibração automática do *baud rate*.
- Transmissão do caractere *Break* de 12bits.

Modo Síncrono (*half-duplex*)

- Seleção da polaridade do *clock* (suportado no modo *Master* e *Slave*).

10.2.1.1 Wake-Up Automático na Recepção de Dado

Durante o modo *Sleep*, tanto a CPU como os periféricos, incluindo o módulo EUSART, são desligados. Consequentemente a EUSART não recebe sinal de *clock*, portanto a recepção do byte não é executada. Se a função *auto-wake-up* estiver habilitada *WUE* (*BAUDCON<1>* = 1), o pino RX é monitorado e quando verificada uma borda de descida nesse pino, o evento de *wake-up* é disparado e o *Flag bit* de interrupção *RCIF* é setado.

Caso a interrupção global esteja habilitada, o programa é desviado para o vetor de interrupção. O bit RCIF é limpo automaticamente através de um comando de leitura do registrador de recepção da EUSART (RCREG).

O bit WUE é limpo automaticamente. Após verificar uma borda de subida no pino RX, o módulo EUSART sai do modo inativo e começa a operar normalmente.

Como o *auto-wake-up* é sensível à borda de descida e o bit WUE é limpo na borda de subida, é recomendado que o primeiro caractere enviado pelo dispositivo RS-232 seja 0x00 (8bits) e 0x000 (12bits) para o barramento LIN, de modo a oferecer tempo suficiente para que o oscilador do sistema possa se estabilizar, antes do módulo EUSART começar a receber os dados.

O bit de status RCIDL (**BAUDCON<6>**) pode ser usado para assegurar que nenhum dado será perdido, pois ele sinaliza se uma operação de recepção está ou não em processo, sendo 1 - Não está em processo e 0 - Em processo. Então se RCIDL = 1, o bit WUE pode ser setado antes de entrar no modo *Sleep*.



A função de *auto-wake-up* está disponível nos modos assíncrono e síncrono Slave com recepção contínua CREN (**RCSTA<4>** = 1).

10.2.1.2 Autodetectão e Calibração do Baud Rate

A autodetectão e calibração do *baud rate* é suportada somente no modo assíncrono e enquanto o *auto-wake-up* está desabilitado WUE (**BAUDCON <1>** = 0). Ela pode ser habilitada através do bit ABDEN (**BAUDCON<0>** = 1), e uma vez habilitada, os registros **SPBRGH:SPBRG** são limpos e utilizados como um contador de 16bits, independente do valor do bit BRG16.

Se a detecção automática do *baud rate* estiver habilitada, o módulo fica aguardando uma borda de subida no pino RX. Ao verificar a primeira borda de subida, o contador inicia a contagem com base na frequência de *clock* do contador BGR (Tabela 10.1) e finaliza ao detectar a quinta borda de subida. Por este motivo o byte recebido deve ser (0x55 = 0b01010101). Na sequência, o bit ABDEN é limpo pelo *hardware* e o *Flag* bit RCIF é setado. A leitura do registro **RCREG** limpa o *Flag* bit RCIF.

Tabela 10.1: Frequência de *clock* do contador BRG (**SPBRGH:SPBRG**).

Resolução do baud rate	Baud rate	Frequência de <i>clock</i> do contador BRG
8bits	baixo	FOSC/512
8bits	alto	FOSC/128
16bits	baixo	FOSC/128
16bits	alto	FOSC/32

Se durante a medida ocorrer um estouro dos registros **SPBRGH:SPBRG** (0xFFFF para 0x0000), o bit de status ABDOVF (**BAUDCON<7>** = 1) é setado, mas o bit ABDEN não é alterado. Logo, o bit ABDOVF deve ser limpo por *software* e a função de detecção de auto *baud rate* deve ser desabilitada (ABDEN = 0) e habilitada (ABDEN = 1) novamente.



O usuário deve ter certeza de que o *baud rate* do byte 0x55 está dentro da faixa da fonte de *clock* do contador BRG.

10.2.1.3 Transmissão de Caractere Break de 12bits

O módulo EUSART tem capacidade de enviar um caractere *Break* (0x000) de 12bits requerido pelo LIN bus padrão.

10.2.1.4 Seleção da Polaridade do Clock

Quando a EUSART está configurada para trabalhar no modo síncrono, o sinal de *clock* no pino CK pode ser configurado como inativo no nível alto (V_{CC}) ou baixo (V_{SS}).

10.3 Funções de Configuração

As funções de configuração da USART/EUSART implementadas no compilador MPLAB® C18 serão apresentadas na sequência, e estão declaradas na biblioteca `uart.h`. Cada comando é representado por três modelos de funções. As indicadas por um número devem ser usadas quando o dispositivo possuir mais de uma USART.

O microcontrolador PIC18F4550 possui apenas uma EUSART, desta forma as funções de configuração são as não numeradas.

10.3.1 Desabilita USART

A função **CloseUSART** desabilita as interrupções, a transmissão e a recepção da USART de dispositivos que contenham apenas um módulo, enquanto a função **ClosexUSART** deve ser usada em dispositivos que possuem mais de uma USART.

Sintaxe

```
CloseUSART()
Close1USART()
Close2USART()
```

Exemplo

```
CloseUSART(); //Desabilita todas as funções relacionadas ao único periférico USART do dispositivo.
Close1USART(); //Desabilita todas as funções relacionadas à USART1 do dispositivo.
Close2USART(); //Desabilita todas as funções relacionadas à USART2 do dispositivo.
```

10.3.2 Habilita USART

A função **OpenUSART** configura o modo de funcionamento do único módulo USART presente no dispositivo, enquanto a função **OpenxUSART** deve ser usada em dispositivos dotados de mais de uma USART.

Sintaxe

```
OpenUSART ( configuração, BRG )
Open1USART ( configuração, BRG )
Open2USART ( configuração, BRG )
```

Sendo:

- **configuração**: configura o modo de funcionamento do módulo USART de acordo com os elementos listados na Tabela 10.4, separados por um '&'.
- **BRG**: seleciona a taxa de comunicação segundo as Tabelas 10.2 (USART) e 10.3 (EUSART).

Tabela 10.2: Taxa de comunicação da USART.

Modo	Velocidade	BRG
Modo assíncrono (<i>baud rate</i> alto)	$V = \frac{F_{osc}}{16 * (BRG + 1)}$	$BRG = \frac{F_{osc}}{V * 16} - 1$
Modo assíncrono (<i>baud rate</i> baixo)	$V = \frac{F_{osc}}{64 * (BRG + 1)}$	$BRG = \frac{F_{osc}}{V * 64} - 1$
Modo síncrono	$V = \frac{F_{osc}}{4 * (BRG + 1)}$	$BRG = \frac{F_{osc}}{V * 4} - 1$

Tabela 10.3: Taxa de comunicação da EUSART.

Modo	Velocidade	BRG
Modo assíncrono <i>baud rate</i> alto e de 8bits <i>baud rate</i> baixo e de 16bits	$V = \frac{F_{osc}}{16 * (BRG + 1)}$	$BRG = \frac{F_{osc}}{V * 16} - 1$
Modo assíncrono <i>baud rate</i> baixo e de 8bits	$V = \frac{F_{osc}}{64 * (BRG + 1)}$	$BRG = \frac{F_{osc}}{V * 64} - 1$
Modo síncrono <i>baud rate</i> alto e de 8bits <i>baud rate</i> baixo e de 16bits	$V = \frac{F_{osc}}{4 * (BRG + 1)}$	$BRG = \frac{F_{osc}}{V * 4} - 1$

Tabela 10.4: Constantes de configuração.

Função	Constante	Descrição
Interrupção na transmissão	USART_TX_INT_ON USART_TX_INT_OFF	Habilita. Desabilita.
Interrupção na recepção	USART_RX_INT_ON USART_RX_INT_OFF	Habilita. Desabilita.
Modo USART	USART_ASYNC_MODE USART_SYNCH_MODE	Modo assíncrono. Modo síncrono.
Quantidade de <i>bits</i> de transmissão/recepção	USART_EIGHT_BIT USART_NINE_BIT	Recebe/transmite 8bits. Recebe/transmite 9bits.
Seleciona o dispositivo como <i>Master</i> ou <i>Slave</i> . <i>Observação:</i> É válido somente no modo Síncrono.	USART_SYNC_SLAVE USART_SYNC_MASTER	Modo <i>Slave</i> . Modo <i>Master</i> .
Modo de recepção	USART_SINGLE_RX USART_CONT_RX	Recepção de um dado. Recepção contínua.
<i>Baud rate</i>	USART_BRGH_HIGH USART_BRGH_LOW	<i>Baud rate</i> alto. <i>Baud rate</i> baixo.

Exemplo

```
#include <p18f452.h> //Arquivo de cabeçalho do PIC18F452
#include <usart.h> //Adiciona a biblioteca contendo as funções da USART.
```

```
// Fosc = 12MHz
// Tciclo = 4/Fosc = 0,333us
#pragma config OSC = HS
```

```

#pragma config WDT = OFF           //Desabilita o Watchdog Timer (WDT).
#pragma config PWRT = ON          //Habilita o Power-up Timer (PWRT).
#pragma config BOR = ON           //Brc:n-out Reset (BOR) habilitado somente no hardware.
#pragma config BORV = 45          //Voltagem do BOR é 4.5V.
#pragma config LVP = OFF          //Desabilita o Low Voltage Program.

void main (void)
{
    TRISA = 0xFF;                //RA0 a RA7 – entrada
    TRISB = 0xFF;                //RB0 - RB7 – entrada.
    TRISC = 0b10111111;           //RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0xFF;                //RD0 - RD7 – entrada.
    TRISE = 0xFF;                //RE0 - RE7 – entrada

    OpenUSART (USART_TX_INT_OFF   //Interrupção de transmissão desabilitada.
              &USART_RX_INT_OFF   //Interrupção de recepção habilitada.
              &USART_SYNCH_MODE  //Modo assíncrono.
              &USART_EIGHT_BIT    //Dado de 8bits.
              &USART_BRGH_HIGH    //Alta velocidade
              ,38);               //Baud rate de 19 200bps.

    putrsUSART ("Teste USART - PIC18F452") //Envia a string para a USART.

    Sleep(); //Coloca o dispositivo no modo Sleep.
}

```

10.3.3 Bits de Configuração do Baud Rate da EUSART

A função **baudUSART** configura as funções adicionais disponibilizadas nos dispositivos com somente um módulo EUSART, enquanto a função **baudxUSART** é usada para configurar a EUSART presente em dispositivos equipados com mais de um módulo.

Sintaxe

baudUSART (configuração)
baud1USART (configuração)
baud2USART (configuração)

Sendo:

- **configuração:** os elementos listados na Tabela 10.5, separados por um '&'.

Tabela 10.5: Bits de configuração do *baud rate* da EUSART.

Função	Constante	Descrição
Estado inativo do <i>clock</i> (Modo síncrono)	BAUD_IDLE_CLK_HIGH BAUD_IDLE_CLK_LOW	Define o nível alto como inativo. Define o nível baixo como inativo.
Gerador de <i>baud rate</i> (BRG)	BAUD_16_BIT_RATE BAUD_8_BIT_RATE	Gerador de 16bits SPBRGH:SPBRG Gerador de 8bits SPBRG
Monitoramento do pino RX	BAUD_WAKEUP_ON BAUD_WAKEUP_OFF	Pino monitorado. Pino não monitorado.
Medida automática do <i>baud rate</i>	BAUD_AUTO_ON BAUD_AUTO_OFF	Habilita. Desabilita.

Exemplo

```
baudUSART (BAUD_8_BIT_RATE           //Gerador de 8bits.  
           &BAUD_AUTO_OFF        //Desabilita o auto baud rate.  
           &BAUD_WAKEUP_OFF);   //Desabilita o auto-wake-up.
```

10.4 Funções de Controle

O compilador MPLAB® C18 disponibiliza um conjunto de funções destinadas ao envio/recepção de um ou mais caracteres e verificação de presença de dado no *buffer* de entrada/saída da USART/EUSART.

Para que elas sejam reconhecidas pelo programa, é necessário adicionar a biblioteca **uart.h**.

10.4.1 Status da Recepção

A função **DataRdyUSART** verifica se há um dado no *buffer* de recepção da única USART de um dispositivo e retorna 1 caso tenha um dado; caso contrário, retorna 0. Essa função deve ser chamada antes de qualquer operação de leitura, para garantir que o conteúdo lido seja efetivamente o dado esperado. O mesmo vale para o **DataRdyxUSART**, no entanto ele é destinado aos dispositivos que contêm mais de um módulo USART.

Sintaxe

```
status = DataRdyUSART ()  
status = DataRdy1USART ()  
status = DataRdy2USART ()
```

Sendo:

- **status**: é um valor binário. 0 - não há dado no *buffer* de recepção e 1 - há um novo dado no *buffer* de recepção.

Exemplo

```
while (!DataRdyUSART ( )); //Aguarda até que seja verificado um novo dado no buffer de recepção.
```

10.4.2 Status da Transmissão

A função **BusyUSART** verifica se a única USART do dispositivo está em um processo de transmissão. Se ela estiver ocupada, a função retorna 1; caso contrário, 0. Essa função deve ser chamada sempre que uma nova transmissão for iniciada, a fim de garantir que todos os dados carregados serão efetivamente enviados. A função **BusyxUSART** desempenha a mesma tarefa, porém é destinada a dispositivos equipados com mais de uma USART.

Sintaxe

```
status = BusyUSART ()  
status = Busy1USART ()  
status = Busy2USART ()
```

Sendo:

- **status**: é um valor binário. 0 - USART não está transmitindo e 1 - USART está em um processo de transmissão.

Exemplo

```
while (BusyUSART( )); //Aguarda até que a USART esteja liberada para uma nova transmissão.
```

10.4.3 Transmissão de Caractere

As funções de envio de caractere, como o próprio nome sugere, transmitem um caractere através da porta serial e devem ser usadas em conjunto com a função **BusyUSART** ou **BusyxUSART** para garantir que todos os dados sejam enviados.

As funções **putcUSART** ou **WriteUSART** são utilizadas em dispositivos que possuem somente um módulo USART. Se o módulo estiver configurado para enviar 9bits, o nono bit deve ser carregado através de **USART_Status.TX_NINE**.

As funções **putcxUSART** e **WritexUSART** são destinadas aos dispositivos dotados de mais de um módulo USART e desempenham a mesma tarefa das funções **putcUSART** ou **WriteUSART**.

Sintaxe

```
putcUSART ( dado )
putc1USART ( dado )
putc2USART ( dado )
WriteUSART ( dado )
Write1USART ( dado )
Write2USART ( dado )
```

Sendo:

- **dado**: é uma constante do tipo **char**, normalmente pertence ao código ASCII. (Apêndice A)



As funções de envio de caractere apresentadas no Capítulo 4 também podem ser usadas para enviar caracteres para a USART.

Exemplo

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include<usart.h> //Adiciona a biblioteca contendo as funções da USART.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)
```

```
void main( ) //Função principal
{
    unsigned char dado_recebido[ ] = {"   "}; //A string deve ser composta por 5 caracteres espaço ''.
    unsigned char n_dado;
```

OSCCONbits IDLEN = 0; //O dispositivo entra no modo Sleep, quando o comando Sleep() for executado.

```
OpenUSART (USART_TX_INT_OFF           //Interrupção de transmissão desabilitada.
           &USART_RX_INT_ON         //Interrupção de recepção habilitada.
           &USART_ASYNC_MODE       //Modo assíncrono.
           &USART_EIGHT_BIT        //Dado de 8bits.
           &USART_BRGH_HIGH        //Alta velocidade
           ,259);                 //Baud rate de 19200bps.
```

```
baudUSART (BAUD_16_BIT_RATE          //Gerador de 16bits.
           &BAUD_AUTO_OFF          //Auto baud rate desabilitado.
           &BAUD_WAKEUP_ON);       //Habilita o auto-wake-up.
```

```
putcUSART ( 'P' ); //Envia o caractere 'P' pelo canal serial padrão.
while (BusyUSART());
putcUSART( 'I' ); //Envia o caractere 'I' pelo canal serial padrão.
```

```

while (BusyUSART());
putcUSART ('C');//Envia o caractere 'C' pelo canal serial padrão
while (BusyUSART());
putrsUSART ("\nDigite seis caracteres "); //Imprime uma string na serial.

// Coloca o dispositivo no modo Sleep.
// O dispositivo sai do modo Sleep, quando for verificada uma borda de descida no pino RX.
// O dado recebido pelo módulo deve ser de preferência 0x00 (caractere NUL).
Sleep ();

while(BAUDCONbits.WUE);//Aguarda a borda de subida no pino RX, antes de efetuar a leitura do registro.

/*Quando o dispositivo sai do modo Sleep, o controlador continua a executar o programa deste ponto em diante.*/
//Efetua a leitura somente para limpar o Flag bit RCIF, pois o primeiro dado é descartado.
getcUSART ();

for (n_dado=0, n_dado<5; n_dado++)//Aguarda até que o módulo EUSART receba cinco caracteres.
{
    while (!DataRdyUSART ( )); //Aguarda até que seja verificado um novo dado no buffer de recepção
    dado_recebido[n_dado] = getcUSART ( ); //Armazena o dado recebido dentro de uma matriz.
}
putsUSART ( dado_recebido ); //Envia a string de caracteres presente na matriz dado_recebido
Sleep ( ); //Coloca o dispositivo no modo Sleep.
}

```

10.4.4 Transmissão de String

As funções para envio de um conjunto de caracteres podem ser classificadas como:

- Envio de *string* de caracteres localizados na memória de dados.
- Envio de *string* de caracteres localizados na memória de programa.



As funções de envio de caractere apresentadas no Capítulo 4 também podem ser usadas para enviar *strings* para a USART.

10.4.4.1 Dados Localizados na Memória de Dados

A função **putsUSART** envia uma *string* de caracteres, incluindo um caractere nulo no final, para a única USART do dispositivo, enquanto a função **putsxUSART** envia a *string* para a USART indicada.

Sintaxe

```

putsUSART ( string )
puts1USART ( string )
puts2USART ( string )

```

Sendo:

- **string**: conjunto de caracteres pertencentes ao código ASCII. (Apêndice A)

Exemplo

```

unsigned char mem_ram[ ] = "Teste PIC18.>"; //Aloca a string na memória de dados.
putsUSART (mem_ram); //Envia a string de caracteres presente na matriz mem_ram.

```

10.4.4.2 Dados Localizados na Memória de Programa

As funções `putrsUSART` e `putrsxUSART` são similares a `putsUSART` e `putsxUSART`, porém os dados estão localizados na memória de programa.

Sintaxe:

```
putrsUSART ( string )
putrs1USART ( string )
putrs2USART ( string )
```

Sendo:

- **string**: conjunto de caracteres pertencentes ao código ASCII. (Apêndice A)

Exemplo

```
putrsUSART ("Teste PIC18\n"); //Envia a string localizada na memória de programa.
```

10.4.5 Recepção de Caractere

As funções `ReadUSART` ou `getcUSART` retornam o conteúdo do *buffer* de recepção da USART, e devem ser usadas em conjunto com a função `DataRdyUSART` para garantir uma correta operação de leitura. Se o módulo estiver configurado para receber 9bits, o nono bit deve ser lido através da estrutura `USART_Status.RX_NINE`. As funções `ReadxUSART` ou `getcxUSART` desempenham a mesma tarefa, porém devem ser usadas somente em dispositivos que possuam mais de um módulo USART.

O *Framing Error* pode ser lido através da estrutura `USART_Status.FRAME_ERROR`, enquanto o *Overrun Error* pode ser lido usando a estrutura `USART_Status.OVERRUN_ERROR`, pois 0 - indica a não existência do erro e 1 - Indica o erro.

Sintaxe:

```
dado = getcUSART ( )
dado = getc1USART ( )
dado = getc2USART ( )
dado = ReadUSART ( )
dado = Read1USART ( )
dado = Read2USART ( )
```

Sendo:

- **dado**: é uma constante do tipo `char`, normalmente pertence ao código ASCII. (Apêndice A)

Exemplo

```
while (!DataRdyUSART ( )); //Aguarda até que seja verificado um novo dado no buffer de recepção.
recebe_dado[1] = ReadUSART ( ); //Armazena o dado recebido no buffer recebe_dado.
```

10.4.6 Recepção de String

A função `getsUSART` realiza a leitura de uma string de caracteres de tamanho predeterminado, vindo do único módulo USART do dispositivo. O controlador fica preso a essa função até que os *n* caracteres sejam recebidos.

A função `getsxUSART` realiza as mesmas tarefas da função `getsUSART`, todavia é usada quando o dispositivo possui mais de um módulo USART.

Sintaxe

```
getsUSART ( string , n )
gets1USART ( string , n )
gets2USART ( string , n )
```

Sendo:

- **string**: é um ponteiro para a matriz de caracteres.
- **n**: número de caracteres que o módulo deve receber.

Exemplo

```
#include<p18f4580.h> //Arquivo de cabeçalho do PIC18F4580.
#include<usart.h> //Adiciona a biblioteca contendo as funções da USART.
#include<stdio.h> //Adiciona a biblioteca padrão de entrada e saída.
```

```
// Fosc = 10MHz
// Tciclo = 4/Fosc = 0,4us
#pragma config OSC = HS

#pragma config WDT = OFF           //Desabilita o Watchdog Timer (WDT).
#pragma config PWRT = ON           //Habilita o Power-up Timer (PWRT).
#pragma config PBAUDEN = OFF        //RB0,1 e 4 configurado como I/O digital
#pragma config LVP = OFF            //Desabilita o Low Voltage Program

void main( ) //Função principal
{
    unsigned char recebe_dado[5];

    TRISEbits.TRISE0 = 0; //RE0 – saída
    LATEdbits.LATE0=0;    //Ex: Desliga o LED

    OpenUSART (USART_TX_INT_OFF      //Interrupção de transmissão desabilitada.
              &USART_RX_INT_OFF      //Interrupção de recepção desabilitada.
              &USART_ASYNCH_MODE     //Modo assíncrono.
              &USART_EIGHT_BIT        //Dado de 8bits.
              &USART_BRGH_HIGH        //Alta velocidade
              ,129);                 //Baud rate de 4800bps.

    baudUSART (BAUD_8_BIT_RATE       //Gerador de 8bits.
              &BAUD_AUTO_OFF         //Auto baud rate desabilitado.
              &BAUD_WAKEUP_OFF);     //Desabilita o auto-wake-up.

    stdout = _H_USART; //Configura o destino de saída da função printf como sendo a USART.

    printf ("\nEnvie cinco caracteres\n"); //Envia uma string para o destino de saída stdout (USART).

    getsUSART (recebe_dado,5 ); //Aguarda a chegada de cinco caracteres vindos da USART.

    //Envia os cinco dados recebidos para o destino de saída stdout (USART).
    printf ("\nDado recebido: %.5s",recebe_dado);

    LATEdbits.LATE0=1;//Ex: Liga o LED.

    Sleep( );//Coloca o dispositivo no modo Sleep.
}
```



10.5 Funções UART Implementadas em Software

O compilador MPLAB® C18 também dispõe de uma biblioteca (**sw_uart.h**), cujas funções permitem implementar uma comunicação serial assíncrona (UART) usando os pinos I/O. Vejamos então essas funções.

10.5.1 Definição das Funções de Atraso

A biblioteca **sw_uart.h** requer que o usuário defina as funções de atraso utilizadas pela UART implementada em software. São elas:

Tabela 10.6: Funções de atraso usadas pelas funções da UART implementada em software.

Nome da função	Atraso
DelayTXBitUART	$\text{Atraso} = \frac{\text{Fosc}}{4 * \text{baud_rate}} + 0.5 - 12 \text{ ciclos}$
DelayRXHalfBitUART	$\text{Atraso} = \frac{\text{Fosc}}{8 * \text{baud_rate}} + 0.5 - 9 \text{ ciclos}$
DelayRXBitUART	$\text{Atraso} = \frac{\text{Fosc}}{4 * \text{baud_rate}} + 0.5 - 14 \text{ ciclos}$

Exemplo

Suponha que Fosc = 20MHz e baud_rate = 9600pbs

#include <delays.h> //Adiciona a biblioteca de funções de delay.

void DelayTXBitUART (void) //Deve gerar um atraso equivalente a 509 ciclos de máquina.

```
{
    Delay100TCYx (5); //500 ciclos
    Delay1TCY ( );//1 ciclo
    Delay1TCY ( );//1 ciclo
}
```

void DelayRXHalfBitUART (void) //Deve gerar um atraso equivalente a 252 ciclos de máquina.

```
{
    Delay100TCYx (2); //200 ciclos
    Delay10TCYx (5); //50 ciclos
    Delay1TCY ( );//1 ciclo
    Delay1TCY ( );//1 ciclo
}
```

void DelayRXBitUART (void) //Deve gerar um atraso equivalente a 507 ciclos de máquina.

```
{
    Delay100TCYx (5); //500 ciclos
    Delay1TCY ( );//1 ciclo
    Delay1TCY ( );//1 ciclo
    Delay1TCY ( );//1 ciclo
    Delay1TCY ( );//1 ciclo
}
```

```

Delay1TCY ( );//1 ciclo
Delay1TCY ( );//1 ciclo
Delay1TCY ( );//1 ciclo
}

```

10.5.2 Configuração da UART em Software

A função **OpenUART** configura os pinos I/O que serão usados pela UART implementada em *software*. Os pinos I/O por padrão são RB4 - TX e RB5 - RX, porém eles podem ser redefinidos alterando os campos da definição **equ** dos arquivos **writuart.asm**, **readuart.asm** e **openuart.asm** localizados em **C:\MCC18\src\pmc_common\SW_UART**.

SWTXD	equ	PORTB	; Porta do pino de transmissão
SWTXDpin	equ	4	; Pino de transmissão
TRIS_SWTXD	equ	TRISB	; Registro TRIS da porta do pino de transmissão
SWRXD	equ	PORTB	; Porta do pino de recepção
SWRXDpin	equ	5	; Pino de recepção
TRIS_SWRXD	equ	TRISB	; Registro TRIS da porta do pino de recepção



Estes arquivos podem ser modificados dentro do projeto, clicando com o botão direito do mouse na pasta "Source Files" e selecionando a opção "Add Files...". Após qualquer modificação, o usuário deve recompilar as rotinas, clicando em **Project → Build All** (Ctrl + F10).

Exemplo:

OpenUART ().

10.5.3 Transmissão de Caractere

A função **putcUART** ou **WriteUART** envia um *dado* para a UART implementada em software

Sintaxe

putcUART (dado)
WriteUART (dado)

Sendo:

- **dado**: é uma constante do tipo **char** que normalmente pertence ao código ASCII. (Apêndice A)

10.5.4 Transmissão de String

A função **putsUART** envia uma *string* de caracteres para a UART implementada em *software*. A *string* deve estar localizada na memória de dados.

Sintaxe

putsUART (string)

Sendo:

- **string**: conjunto de caracteres pertencentes ao código ASCII. (Apêndice A)

10.5.5 Recepção de Caractere

A função **ReadUART** ou **getcUART** retorna o dado lido pelo pino de recepção de dado, definido pela USART implementada em *software*.

Sintaxe

```
dado = getcUART ()
dado = ReadUART ()
```

Sendo:

- **dado**: é uma constante do tipo **char** que normalmente pertence ao código ASCII. (Apêndice A)

10.5.6 Recepção de String

A função **getsUART** realiza a leitura de uma *string* de caracteres de tamanho predeterminado, lidos pelo pino de entrada da USART implementada em *software*.

Sintaxe

```
getsUART ( string , n )
```

Sendo:

- **string**: é um ponteiro para a matriz de caracteres.
- **n**: número de caracteres que o módulo deve receber.

Exemplo

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <sw_uart.h> //Adiciona a biblioteca de funções da USART implementada em software
#include <delays.h> //Adiciona a biblioteca de funções de delay.
```

//Fosc = 20MHz e baud_rate = 9600bps. Deve gerar um atraso equivalente a 509 ciclos de máquina.
void DelayTXBitUART (void)

```
{
    Delay100TCYx (5); //500 ciclos
    Delay1TCY ( );//1 ciclo
    Delay1TCY ( );//1 ciclo
}
```

//Fosc = 20MHz e baud_rate = 9600bps. Deve gerar um atraso equivalente a 252 ciclos de máquina.
void DelayRXHalfBitUART (void)

```
{
    Delay100TCYx (2); //200 ciclos
    Delay10TCYx (5); //50 ciclos
    Delay1TCY ( );//1 ciclo
    Delay1TCY ( );//1 ciclo
}
```

//Fosc = 20MHz e baud_rate = 9600bps. Deve gerar um atraso equivalente a 507 ciclos de máquina.
void DelayRXBitUART (void)

```

{
    Delay100TCYx (5); //500 ciclos
    Delay1TCY ( );//1 ciclo
    Delay1TCY ( );//1 ciclo
}

//Configuração do oscilador:
//Suponha que um cristal de 20MHz esteja sendo usado
#pragma config FOSC = HS //Clock externo >= 4MHz.
//A frequência do oscilador (Fosc) é igual a frequência do clock externo, logo, Fosc=20MHz
//Um ciclo de máquina Tmaq = 4/Fosc, Tmaq = 4/20 = 0,2us.
#pragma config CPUDIV = OSC1_PLL2 //Clock externo dividido por 1.

#pragma config MCLRE = ON           //Habilita o pino MCLR.
#pragma config WDT = OFF            //Desabilita o Watchdog Timer (WDT).
#pragma config PWRT = ON            //Habilita o Power-up Timer (PWRT).
#pragma config BOR = ON             //Brown-out Reset (BOR) habilitado somente no hardware
#pragma config BORV = 1              //Voltagem do BOR é 4,33V.
#pragma config WRTD = OFF            //Desabilita proteção de escrita na EEPROM.
#pragma config STVREN = ON           //Habilita o Reset por Stack full/underflow.
#pragma config PBADEN = OFF          //RB0,1,2,3 e 4 configurado como I/O digital.
#pragma config LVP = OFF             //Desabilita o Low Voltage Program

void main( ) //Função principal
{
    unsigned char recebe_dado[5];
    char string_ram[]="ENVIE UM CARACTERE: ";

    OpenUART ( ); //Configura os pinos I/O para a UART. RXD = RB5 e TXD = RB4.
    putsUART (string_ram); //Envia uma string para a UART.
    putcUART (getcUART ( ));//Retorna o caractere recebido
    while(1); //Looping infinito.
}

```

10.6 Projeto

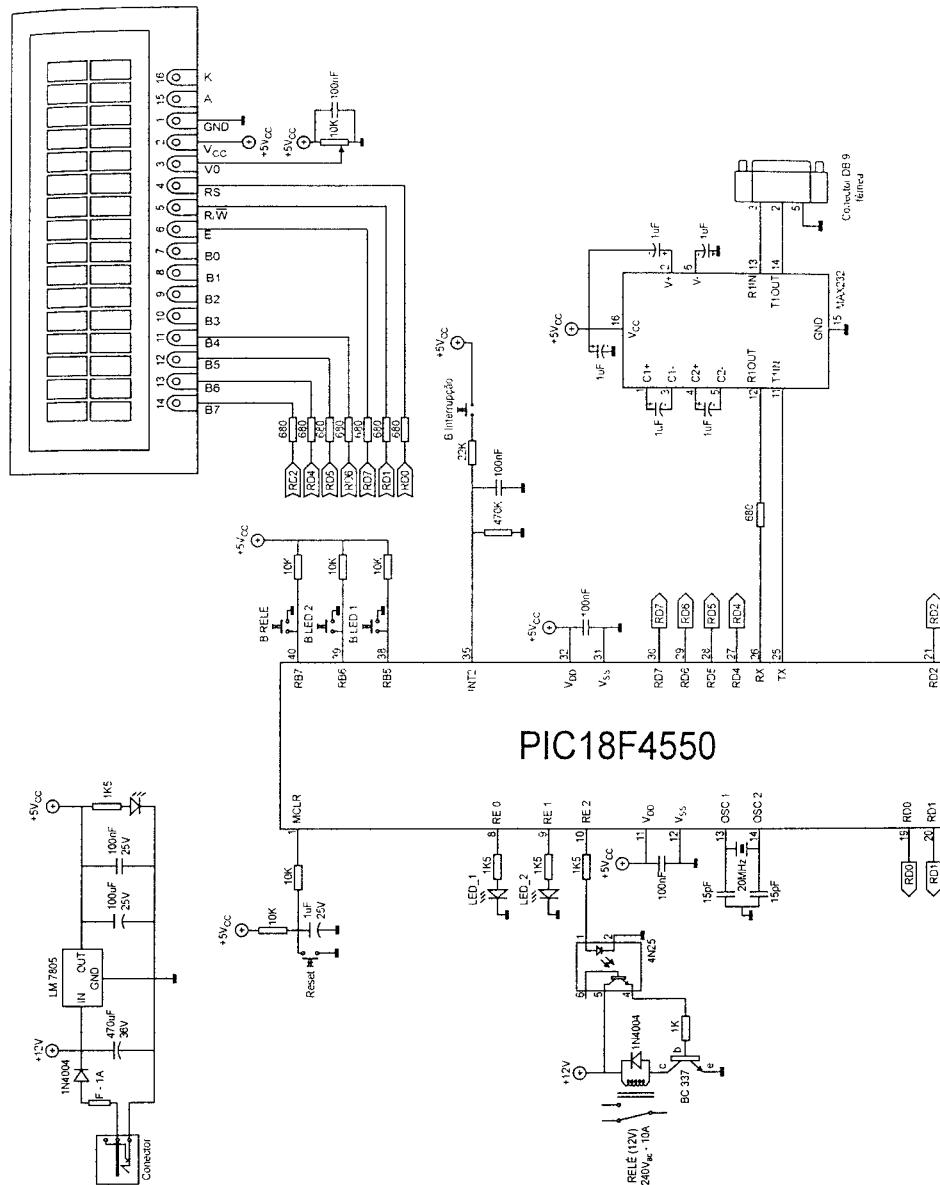


Figura 10.5: Circuito de comunicação serial RS-232.

Código

```
***** Projeto Capítulo 10 - COMUNICAÇÃO RS-232 ****
**
** Este código mostra no visor do display 2x16, os status das saídas
** As saídas, LED e RELÉ, são controladas via porta serial
** LED_1: 'a' - Liga e 'b' - Desliga
** RELÉ: 'c' - Liga e 'd' - Desliga
**
** Autor: Alberto Noboru Miyadaira
***** */

#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <adc.h>      //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <delays.h>    //Adiciona a biblioteca de funções de delay.
#include<usart.h>     //Adiciona a biblioteca contendo as funções da USART.
#include "C:\pic18\ biblioteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550 (Capítulo 6)

void ISR_alta_prioridade(void); //Protótipo da função de interrupção.

unsigned char status_led = 0, status_rele = 0;

#define RELE PORTEbits.RE2 //Define outro nome para a estrutura.
#define LED_1 PORTEbits.RE0 //Define outro nome para a estrutura.

#pragma code int_alta=0x08 //Vetor de interrupção de alta prioridade, ou padrão.
void int_alta (void)
{
    _asm GOTO ISR_alta_prioridade _endasm //Desvia o programa para a função ISR_alta_prioridade.
}
#pragma code

#pragma interrupt ISR_alta_prioridade
void ISR_alta_prioridade(void)
{
    switch(getcUSART( )) //Lê o dado recebido pela serial, e verifica em qual dos casos ele se enquadra
    {
        case 'a': status_led=1; break;
        case 'b': status_led=0; break;
        case 'c': status_rele=1; break;
        case 'd': status_rele=0; break;
    }
}

RELE = status_rele; //Seta o pino de acordo com o valor contido na variável status_rele.
LED_1 = status_led; //Seta o pino de acordo com o valor contido na variável status_led.
}

void main (void)
{
    TRISA = 0b00011111;          // RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11100111;          // RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111;          // RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0b00000000;          // RD0 a RD7 – saída.
```

```

TRISE = 0b00000000;           // RE0 a RE2 -- saída.

//Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital (Capítulo 13)
//Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h.
CloseADC ( );                //Requer a biblioteca adc.h.

```

```

PORTD = 0x00; //Coloca a porta D em 0V.
PORTE = 0x00; //Coloca a porta E em 0V.

```

```

OpenUSART (USART_TX_INT_OFF      //Interrupção de transmissão desabilitada.
           &USART_RX_INT_ON    //Interrupção de recepção habilitada.
           &USART_ASYNCH_MODE //Modo assíncrono.
           &USART_EIGHT_BIT    //Dados de 8bits.
           &USART_BRGH_HIGH   //Alta velocidade
           ,129);             //Baud rate de 9600bps

```

```

baudUSART (BAUD_8_BIT_RATE      //Gerador de 8bits.
           &BAUD_AUTO_OFF     //Auto baud rate desabilitado.
           &BAUD_WAKEUP_OFF); //Desabilita o auto-wake-up.

```

```

lcd_inicia(0x28, 0x0F, 0x06); //Inicializa o display LCD alfanumérico com quatro linhas de dados.
lcd_LD_cursor (0); //Desliga o cursor.

```

```
IPR1bits.RCIP = 1; //interrupção de recepção da EUSART com prioridade alta.
```

```

RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo.
INTCONbits.GIEH = 1; //Habilita todas as interrupções de alta prioridade.
INTCONbits.GIEL = 1; //Habilita todas as interrupções de baixa prioridade.

```

```

while ( 1 ) //Looping infinito
{
    lcd_posicao (1,1); // Desloca o ponteiro para a L=1 e C=1.

    if (status_led == 0) //Verifica o status do LED.
    { imprime_string_lcd("LED_1 - Desligado "); } //Imprime o status do LED_1.
    else
    { imprime_string_lcd("LED_1 - Ligado. "); } //Imprime o status do LED_1.

    lcd_posicao (2,1); // Desloca o ponteiro para a L=2 e C=1.

    if (status_rele == 0) //Verifica o status do RELE.
    { imprime_string_lcd("RELE - Desligado "); } //Imprime o status do RELÉ.
    else
    { imprime_string_lcd("RELE - Ligado. "), } // Imprime o status do RELÉ.

    Delay10KTCYx (100); //Se Fosc = 20MHz. Gera um atraso de 200ms.
}

```

11

TIMERs e Watchdog Timer (WDT)

11.1 TIMERs

O microcontrolador PIC18F4550 possui quatro TIMERs, sendo TIMER 0, TIMER 1, TIMER 2 e TIMER 3, que são aplicados fundamentalmente como temporizadores ou contadores. Eles podem ser configurados para serem incrementados por um sinal de *clock* interno (ciclo de máquina), para o primeiro caso ou simplesmente por uma fonte externa acoplada a um pino específico para o segundo.

Quando configurados como temporizadores, como o próprio nome sugere, são usados para realizar contagem de tempo. No modo contador, eles contam a quantidade de vezes que um determinado evento ocorre, baseado na borda de subida ou descida do sinal externo.

Os TIMERs possuem um bloco *prescaler*, e em alguns casos, um *postscaler*. O *prescaler* tem a função de definir o número de vezes que um determinado evento deve ocorrer, antes de o registro ser incrementado. Por exemplo, suponha que o módulo TIMER reconheça um evento na borda de subida do sinal de entrada; logo, se o *prescaler* associado a ele for igual a 2, significa que a cada duas bordas de subida o bloco fornece uma. Normalmente é o sinal de saída do bloco *prescaler* que incrementa o registro do TIMER.

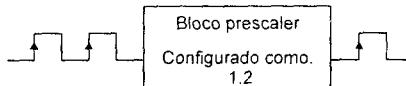


Figura 11.1: Bloco *prescaler* configurado como 1:2.

O outro bloco é o *postscaler*, que funciona como um contador de sinais enviados por um bloco comparador, o qual está presente nos módulos TIMER 2 e TIMER 4, cuja função é comparar o valor do registro **TMRx** com um valor de referência (**PRx**), e enviar um sinal para o bloco, caso seja verificada a igualdade. O valor definido no *postscaler* determina a quantidade de sinais que o bloco deve receber, para que o *Flag bit* de interrupção seja setado.

Por exemplo, se o *postscaler* estiver configurado como 1:5, somente no quinto sinal enviado pelo bloco comparador o *Flag bit* de interrupção será setado.

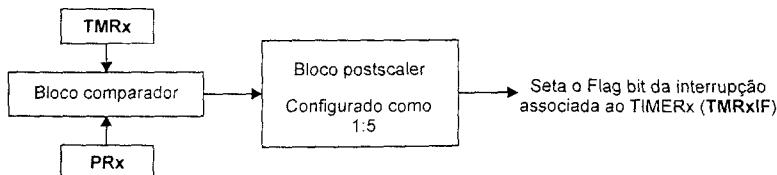


Figura 11.2: Bloco *postscaler* configurado como 1:5.

11.1.1 Características dos TIMERS do PIC18F4550

11.1.1.1 TIMER 0

- Pode operar como contador ou temporizador de 8bits (0 - 255) ou 16bits (0 - 65535).
- A fonte de *clock* pode ser o pino de entrada T0CKI ou o *clock* do sistema (Fosc/4).

11.1.1.2 TIMER 1

- Pode operar como contador (síncrono ou assíncrono) ou temporizador de 16bits (0 – 65535).
- A fonte de *clock* pode ser o pino de entrada T13CKI ou o *clock* do sistema (Fosc/4).
- A operação de escrita/leitura do registro **TMR1** pode ser realizada por meio de duas operações de 8bits, ou por uma única operação de 16bits. A operação de 8bits não é tão precisa quanto a de 16bits, pois os registros **TMR1L** e **TMR1H** são lidos/escritos um por vez. No modo de 16bits, o comando de leitura/leitura do **TMR1L** automaticamente atualiza o registro **TMR1H** com o valor do **TMR1 high byte**, resultando uma leitura dos 16bits do **TMR1** em uma única operação.
- Possui um circuito oscilador *low-power*, localizado entre os pinos T10SI (entrada) e T10SO (saída), destinado a cristais de 32KHz (veja o Capítulo 5). Esse oscilador pode operar em dois níveis distintos de consumo de energia, cuja seleção pode ser feita pelo *bit* de configuração LPT1OSC (veja o Capítulo 6).
- Pode ser utilizado pelo módulo CCP como base de tempo para o Capture ou Compare.



Se o módulo TIMER 1 estiver configurado para receber sinal de *clock* externo, o sinal de saída do estágio *prescaler* pode ser sincronizado ou não, com a fase de *clock* interno, antes de incrementar os registros do TIMER 1. Se a sincronização estiver habilitada, a entrada de *clock* externo é sincronizada com a fase de *clock* interno, e o incremento ocorre na borda Q4:Q1; caso contrário, a contagem é feita assincronamente. Esta última opção é usada quando a CPU está no modo Sleep.

11.1.1.3 TIMER 2

- Pode operar como temporizador de 8bits (0 - 255).
- A fonte de *clock* do sistema (Fosc/4).
- Utilizado pelo módulo CCP como base de tempo para o PWM.
- Pode ser usado como fonte de *clock* para o módulo MSSP, operando no modo SPI.
- Esse módulo conta com um *period register* (**PR2**) de 8bits que é utilizado como um "limitador de contagem" do **TMR2**. Diferentemente dos TIMERS comentados até o presente momento, o **TMR2** é comparado com o valor de **PR2** a cada ciclo de *clock*, e quando ambos forem iguais, o comparador envia um sinal que reinicia o **TMR2** e incrementa o *postscaler*.

11.1.1.4 TIMER 3

- Pode operar como contador (síncrono ou assíncrono) ou temporizador de 16bits (0 - 65535).
- A fonte de *clock* pode ser o pino de entrada T13CKI ou o *clock* do sistema (Fosc/4).
- Do mesmo modo que o TIMER 1, a operação de escrita/leitura do registro **TMR3** pode ser realizada por meio de duas operações de 8bits ou por uma única operação de 16bits.
- Pode ser utilizado pelo módulo CCP como base de tempo para o Capture ou Compare.



Se a fonte de *clock* do módulo TIMER 3 for o sinal de entrada no pino T13CKI, seu comportamento é idêntico ao apresentado pelo TIMER 1.

11.1.2 Funções de TIMER

As funções de TIMER que serão apresentadas a seguir podem ser usadas em qualquer microcontrolador PIC18, desde que ele possua o TIMER especificado. Elas estão declaradas dentro da biblioteca **timers.h**.

11.1.2.1 Desabilita TIMER

A função **CloseTimerx** desabilita o TIMER especificado.

Sintaxe

```
CloseTimer0()
CloseTimer1()
CloseTimer2()
CloseTimer3()
CloseTimer4()
```

Exemplo

```
CloseTimer0(); //Desabilita o TIMER 0.
CloseTimer1(); //Desabilita o TIMER 1.
```

11.1.2.2 Habilita TIMER 0

A função **OpenTimer0** habilita e configura o TIMER 0.

Sintaxe

```
OpenTimer0( configuração )
```

Sendo:

- configuração:** são as constantes listadas na Tabela 11.1, separadas por um '&'

Tabela 11.1: Constantes de configuração do TIMER 0.

Função	Constante	Descrição
Interrupção de TIMER 0	TIMER_INT_ON TIMER_INT_OFF	Habilita. Desabilita.
Tamanho do contador	T0_8BIT T0_16BIT	8bits. 16bits.
Fonte de <i>clock</i>	T0_SOURCE_EXT T0_SOURCE_INT	Fonte de <i>clock</i> externa. Fonte de <i>clock</i> interna.
Borda de incremento <i>Observação:</i> Válido somente para fonte de <i>clock</i> externa.	T0_EDGE_FALL T0_EDGE_RISE	Borda de subida. Borda de descida.
Valor de <i>prescaler</i>	T0_PS_1_1 T0_PS_1_2 T0_PS_1_4 T0_PS_1_8 T0_PS_1_16 T0_PS_1_32 T0_PS_1_64 T0_PS_1_128 T0_PS_1_256	1:1 <i>prescaler</i> . 1:2 <i>prescaler</i> . 1:4 <i>prescaler</i> . 1:8 <i>prescaler</i> . 1:16 <i>prescaler</i> . 1:32 <i>prescaler</i> . 1:64 <i>prescaler</i> . 1:128 <i>prescaler</i> . 1:256 <i>prescaler</i> .

Exemplo

```
OpenTimer0(TIMER_INT_ON           //Habilita a interrupção do TIMER 0.
          &T0_8BIT            //Operação de 8bits.
          &T0_SOURCE_EXT      //Seleciona a fonte de clock externa (pino T0CKI).
          &T0_EDGE_RISE       //Valida o sinal na borda de subida
          &T0_PS_1_64);       //Prescaler igual a 64 (1:64).
```

11.1.2.3 Habilita TIMER 1

A função **OpenTimer1** habilita e configura o TIMER 1.

Sintaxe

```
OpenTimer1 ( configuração )
```

Sendo:

- configuração:** são as constantes listadas na Tabela 11.2, separadas por um '&'.

Tabela 11.2: Constantes de configuração do TIMER 1.

Função	Constante	Descrição
Interrupção de TIMER 1	TIMER_INT_ON TIMER_INT_OFF	Habilita. Desabilita.
Tamanho do contador	T1_8BIT_RW T1_16BIT_RW	8bits. 16bits.
Fonte de clock	T1_SOURCE_EXT T1_SOURCE_INT	Fonte de clock externa. Fonte de clock interna.
Valor de prescaler	T1_PS_1_1 T1_PS_1_2 T1_PS_1_4 T1_PS_1_8	1:1 prescaler. 1:2 prescaler. 1:4 prescaler. 1:8 prescaler.
Circuito oscilador	T1_OSC1EN_ON T1_OSC1EN_OFF	Habilita. Desabilita.
Sincronismo da entrada de clock externa	T1_SYNC_EXT_ON T1_SYNC_EXT_OFF	Habilita. Desabilita.

Exemplo

```
OpenTimer1(TIMER_INT_ON           //Habilita a interrupção do TIMER 1.
          &T1_16BIT_RW        //Operação de 16bits.
          &T1_SOURCE_INT      //Seleciona a fonte de clock interna (Fosc/4).
          &T1_OSC1EN_OFF      //Desabilita o oscilador.
          &T1_SYNC_EXT_OFF    //Modo assíncrono.
          &T1_PS_1_4);         //Prescaler igual a 4 (1:4).
```

11.1.2.4 Habilita TIMER 2

A função **OpenTimer2** habilita e configura o TIMER 2.

Sintaxe

```
OpenTimer2 ( configuração )
```

Sendo:

- **configuração:** são as constantes listadas na Tabela 11.3, separadas por um '&'.

Tabela 11.3: Constantes de configuração do TIMER 2.

Função	Constante	Descrição
Interrupção de TIMER 2	TIMER_INT_ON TIMER_INT_OFF	Habilita. Desabilita.
Valor de prescaler	T2_PS_1_1 T2_PS_1_4 T2_PS_1_16	1:1 prescaler. 1:4 prescaler. 1:16 prescaler.
Valor de postscaler	T2_POST_1_1 T2_POST_1_2 ... T2_POST_1_15 T2_POST_1_16	1:1 postscaler. 1:2 postscaler. ... 1:15 postscaler. 1:16 postscaler.

Exemplo

```
OpenTimer2(TIMER_INT_ON      //Habilita a interrupção do TIMER 2.  
          &T2_POST_1_10    //Postscaler igual a 10 (1:10)  
          &T2_PS_1_16);   //Prescaler igual a 16 (1:16).
```

11.1.2.5 Habilita TIMER 3

A função **OpenTimer3** habilita e configura o TIMER 3.

Sintaxe

```
OpenTimer3( configuração )
```

Sendo:

- **configuração:** são as constantes listadas na Tabela 11.4, separadas por um '&'.

Tabela 11.4: Constantes de configuração do TIMER 3.

Função	Constante	Descrição
Interrupção de TIMER 3	TIMER_INT_ON TIMER_INT_OFF	Habilita. Desabilita.
Tamanho do contador	T3_8BIT_RW T3_16BIT_RW	8bits. 16bits.
Fonte de clock	T3_SOURCE_EXT T3_SOURCE_INT	Fonte de clock externa. Fonte de clock interna.
Valor de prescaler	T3_PS_1_1 T3_PS_1_2 T3_PS_1_4 T3_PS_1_8	1:1 prescaler. 1:2 prescaler. 1:4 prescaler. 1:8 prescaler.
Sincronismo da entrada de clock externa	T3_SYNC_EXT_ON T3_SYNC_EXT_OFF	Habilita. Desabilita.

Exemplo

```
OpenTimer3(TIMER_INT_OFF           //Desabilita a interrupção do TIMER 3
           &T3_8BIT_RW          //Operação de 8bits.
           &T3_SOURCE_INT        //Seleciona a fonte de clock interna (Fosc/4).
           &T3_SYNC_EXT_ON       //Modo síncrono.
           &T3_PS_1_1);         //Prescaler igual a 1 (1.1).
```

11.1.2.6 Habilita TIMER 4

A função **OpenTimer4** habilita e configura o TIMER 4.

Sintaxe

```
OpenTimer4( configuração )
```

Sendo:

- configuração:** são as constantes listadas na Tabela 11.5, separadas por um '&'.

Tabela 11.5: Constantes de configuração do TIMER 4

Função	Constante	Descrição
Interrupção de TIMER 4	TIMER_INT_ON TIMER_INT_OFF	Habilita. Desabilita.
Valor de prescaler	T4_PS_1_1 T4_PS_1_4 T4_PS_1_16	1:1 prescaler. 1:4 prescaler. 1:16 prescaler.
Valor de postscaler	T4_POST_1_1 T4_POST_1_2 ... T4_POST_1_15 T4_POST_1_16	1:1 postscaler. 1:2 postscaler. ... 1:15 postscaler. 1:16 postscaler

Exemplo

```
OpenTimer4(TIMER_INT_OFF           //Desabilita a interrupção do TIMER 4.
           &T4_POST_1_16      //Postscaler igual a 16 (1:16).
           &T4_PS_1_4);        //Prescaler igual a 4 (1:4).
```

11.1.2.7 Operação de Leitura

A função **ReadTimerx** retorna o valor do respectivo TIMER.

Sintaxe

```
valor_16bits = ReadTimer0()
valor_16bits = ReadTimer1()
valor_8bits = ReadTimer2()
valor_16bits = ReadTimer3()
valor_8bits = ReadTimer4()
```

Sendo:

- valor_16bits:** valor do tipo **int** (16bits).
- valor_8bits:** valor do tipo **char** (8bits).

Exemplo

```

tempo_8bits = ReadTimer2();
//Lê o registro TMR2
tempo_16bits = ReadTimer1();
//Lê os registros TMR1H:TMR1L
tempo_16bits = ReadTimer3();
//Lê os registros TMR3H:TMR3L

```

11.1.2.8 Operação de Escrita

A função **WriteTimerx** escreve um valor dentro do respectivo TIMER.

Sintaxe

```

WriteTimer0 (valor_16bits)
WriteTimer1 (valor_16bits)
WriteTimer2 (valor_8bits)
WriteTimer3 (valor_16bits)
WriteTimer4 (valor_8bits)

```

Sendo:

- **valor_16bits**: valor do tipo **int (16bits)**.
- **valor_8bits**: valor do tipo **char (8bits)**.

Exemplo

```

WriteTimer0 (0x00);
WriteTimer1 (20332);
WriteTimer4 (0x20);

```

11.1.2.9 Seleção do TIMER para o Módulo CCP

A função **SetTmrCCPSrc** habilita o TIMER para o módulo CCP.

Sintaxe

```
SetTmrCCPSrc ( configuração )
```

Sendo:

- **configuração**: são as constantes listadas na Tabela 11.6.

Tabela 11.6: Constantes de configuração do TIMER para o módulo CCP.

Função	Constante	Descrição
Fonte de <i>clock</i> para o módulo CCP <i>Observação</i> : Dispositivos com um ou dois módulos CCP.	T3_SOURCE_CCP T1_CCP1_T3_CCP2 T1_SOURCE_CCP	TIMER 3 para o CCP1 e CCP2. TIMER 1 para o CCP1. TIMER 3 para o CCP2. TIMER 1 para o CCP1 e CCP2.
Fonte de <i>clock</i> para o módulo CCP <i>Observação</i> : Dispositivos com mais de dois módulos CCP.	T34_SOURCE_CCP T12_CCP12_T34_CCP345 T12_CCP1_T34_CCP2345 T12_SOURCE_CCP	TIMER 3 e 4 para todos os CCPs. TIMER 1 e 2 para o CCP1 e CCP2. TIMER 3 e 4 para o CCP3 a CCP5. TIMER 1 e 2 para o CCP1. TIMER 3 e 4 para o CCP2 a CCP5. TIMER 1 e 2 para todos os CCPs.

Exemplo

```
SetTmrCCPSrc (T1_SOURCE_CCP);
```

11.2 Watchdog Timer (WDT)

O *Watchdog Timer* (WDT) é um temporizador configurável, aplicado em *hardware*, cujo objetivo é evitar que o processador fique preso dentro de uma determinada rotina, por um período de tempo superior ao especificado, reiniciando o dispositivo, caso esse evento venha a ocorrer. Ele é essencial na programação de microcontroladores ou microprocessadores, pois evita que o programa "trave".

O módulo WDT do microcontrolador PIC18F4550 é alimentado por uma fonte de *clock* interna (INTRC) de 32KHz, a qual é automaticamente habilitada se o WDT for ativado. Esse módulo pode ser habilitado setando o bit SWDTEN (**WDTCON<0>**), ou através do bit de controle WDTEN (**#pragma config WDTEN = ON**)

Os possíveis valores de tempo para o WDT estão listados na tabela a seguir.

Tabela 11.7: Valores de tempo para o WDT do PIC18F4550.

| Tempo do WDT |
|--------------|--------------|--------------|--------------|--------------|--------------|
| 4ms | 8ms | 16ms | 32ms | 64ms | 128ms |
| 256ms | 512ms | 1.024ms | 2.048ms | 4.096ms | 8.192ms |
| 16 384ms | 32.768ms | 65.536ms | 131.072ms | | |



As instruções assembly **CLRWD**T (em C - **ClrWdt()**) e **SLEEP** (em C - **Sleep()**), quando executadas, limpam os contadores de WDT e Postscaler.

11.2.1 Função de Reinício do Contador de WDT

Sempre que a função **ClrWdt()** é executada, os contadores de WDT e Postscaler são reiniciados. Essa função deve ser constantemente chamada durante o programa, de modo a evitar que o dispositivo seja reiniciado inesperadamente.

Sintaxe

ClrWdt()

Exemplo

/* Reinicia o microcontrolador, se o sinal presente no pino RB5 for equivalente a zero, durante um período de tempo superior a 512ms. */

```
#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <adc.h> //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <delays.h> //Adiciona a biblioteca de funções de delay.
```

```
// Fosc = 20MHz
// Tciclo = 4/Fosc = 0,2us
#pragma config FOSC = HS
#pragma config CPUDIV = OSC1_PLL2

#pragma config MCLRE = ON          //Habilita o pino MCLRE.
#pragma config WDT = ON           //Habilita o Watchdog Timer (WDT).
#pragma config WDTPS = 128         //Período do WDT equivalente a 512ms.
#pragma config PWRT = ON          //Habilita o Power-up Timer (PWRT).
#pragma config BOR = ON           //Brown-out Reset (BOR) habilitado somente no hardware
#pragma config BORV = 1            //Voltagem do BOR é 4,33V.
#pragma config WRTD = OFF          //Desabilita proteção de escrita na EEPROM.
#pragma config STVREN = ON          //Habilita o Reset por Stack full/underflow.
#pragma config PBADEN = OFF        //RB0,1,2,3 e 4 configurado como I/O digital.
#pragma config LVP = OFF           //Desabilita o Low Voltage Program.
```

```

#pragma config LPT1OSC = ON           //TIMER 1 programado para operar no modo low-power.

#define B_LED_1 PORTBbits.RB5        //Define outro nome para a estrutura.
#define LED_1 PORTEbits.RE0         //Define outro nome para a estrutura.

void main( )
{
    TRISA = 0b00011111; // RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11100111; // RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111; // RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0b00000000; // RD0 a RD7 – saída.
    TRISE = 0b00000000; // RE0 a RE2 – saída.

    //Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital. (Capítulo 13)
    //Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
    OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h.
    CloseADC (); //Requer a biblioteca adc.h.

    PORTD = 0x00; //Coloca a porta D em 0V.
    PORTE = 0x00; //Coloca a porta E em 0V.

    LED_1 = 0://Desliga o LED_1.

    Delay10KTCYx (250); //Gera um delay de 500 milissegundos. 250*10.000*0,2us= 500ms
    ClrWdt(); //Reinicia o WDT.
    Delay10KTCYx (250); //Gera um delay de 500 milissegundos. 250*10.000*0,2us= 500ms
    ClrWdt(); //Reinicia o WDT.
    Delay10KTCYx (250); //Gera um delay de 500 milissegundos. 250*10.000*0,2us= 500ms
    ClrWdt(); //Reinicia o WDT.

    LED_1 = 1;//Liga o LED_1.

    while(1)//laço infinito
    {
        /* Enquanto o pino RB5 estiver em nível baixo (pressionado), o controlador fica preso à rotina e deixa de reiniciar o
        WDT. Se o pino ficar pressionado por um período de tempo superior a 512ms, o microcontrolador será reiniciado. */
        while ( B_LED_1 == 0 );

        ClrWdt(); //Reinicia o WDT.
    }
}

```

11.3 Projeto

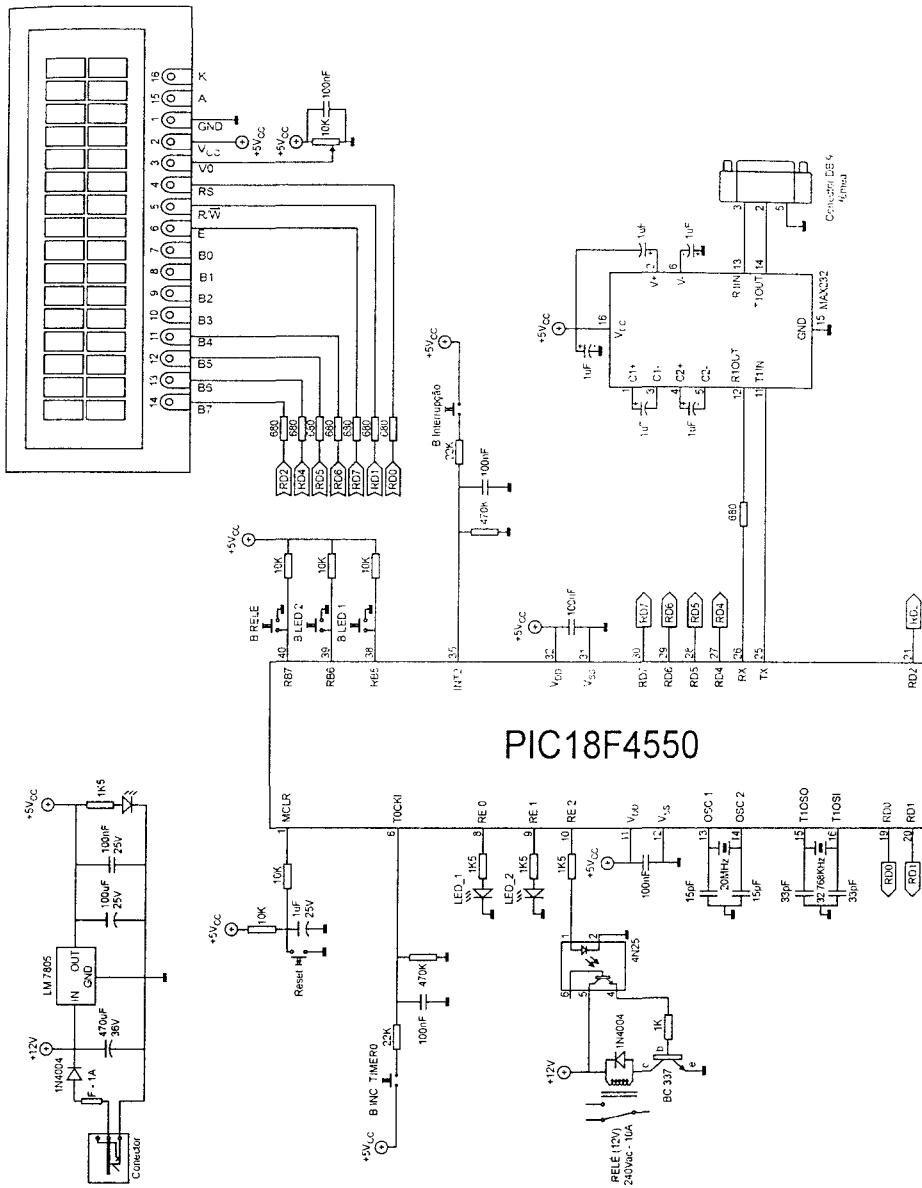


Figura 11.3: Circuito contador TIMER1.

Código

```
*****
***** Projeto Capítulo 11 – TIMER0 *****
**
** Este programa mostra no visor do display 2x16 o valor do contador do TIMER0.
** Quando o valor do contador for igual a 10, então o LED_1 será ligado.
** O LED_1 é desligado quando o valor do contador for igual a 15.
**
** Autor: Alberto Noboru Miyadaira
***** */

#include <p18f4550.h>           //Arquivo de cabeçalho do PIC18F4550.
#include <stdio.h>               //Adiciona a biblioteca padrão de entrada e saída.
#include <delays.h>              //Adiciona a biblioteca de funções de delay.
#include <adc.h>                 //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <timers.h>              //Adiciona a biblioteca de funções de TIMER.
#include "C:\pic18\ biblioteca_Lcd_2x16.h" //Biblioteca contendo as funções do LCD.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550 (Capítulo 6)

#define D_LED_1 PORTEbits.RE0 = 0; //Define um nome para a macro.
#define L_LED_1 PORTEbits.RE0 = 1; //Define um nome para a macro.

unsigned char buffer[14];

void main (void)
{
    TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11001111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0b00000000; //RD0 a RD7 – saída.
    TRISE = 0b00000000; //RE0 a RE2 – saída.

    OpenTimer0(TIMER_INT_OFF          //Desabilita a interrupção do TIMER 0.
              &T0_16BIT           //Modo de 16bits.
              &T0_SOURCE_EXT      //Seleciona a fonte externa (T0CKI).
              &T0_EDGE_FALL       //Incrementa no borda de descida do sinal.
              &T0_PS_1_1);        //Prescaler igual a 1 (1:1).

    //Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital. (Capítulo 13)
    //Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
    OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h.
    CloseADC ( );                //Requer a biblioteca adc.h.

    PORTD = 0x00; //Coloca a porta D em 0V.
    PORTE = 0x00; //Coloca a porta E em 0V.

    lcd_inicia(0x28, 0x0F, 0x06); //Inicializa o display LCD alfanumérico com quatro linhas de dados.
    lcd_LD_cursor (0); //Desliga o cursor.

    D_LED_1 //Desliga o LED_1.

    while (1) //Looping infinito.
    {
        lcd_posicao (1,1); //Desloca o ponteiro para a L=1 e C=1.
        imprime_string_lcd ("Contador-TIMER1"); //Imprime uma string no display.
    }
}
```

```

lcd_posicao(2,1); // Desloca o ponteiro para a L=2 e C=1
sprintf(buffer, "TIMER0 = %05u", ReadTimer0()); //Envia a string formatada para o buffer.
imprime_buffer_lcd(buffer,14); //Coloca no visor, os 14 primeiros elementos do buffer.

if (ReadTimer0() == 10) //Se o pino T0CKI for pressionado 10 vezes, o LED_1 é ligado.
{ L_LED_1; } //Liga o LED_1.
if (ReadTimer0() == 15) //Se o pino T0CKI for pressionado 15 vezes, o LED_1 é desligado
{ D_LED_1; } //Desliga o LED_1.
Delay10KTCYx(100); //Gera um delay de 200 milissegundos. 100*10.000*0,2us= 200ms
}
}

```

```

*****
***** Projeto Capítulo 11 – TIMER1 *****
**
**          Real Time Clock (RTC)
**
** Observação. A frequência do cristal usado no circuito oscilador do TIMER 1 é 32.768Hz
**          O oscilador deve estar configurado como low-power.
**          A contagem deve ser assíncrona, a fim de obter maior precisão.
**
** Autor: Alberto Noboru Miyadaira
*****

```

```

#include<p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include<stdio.h>    //Adiciona a biblioteca padrão de entrada e saída.
#include <delays.h>   //Adiciona a biblioteca de funções de delay.
#include <timers.h>   //Adiciona a biblioteca de funções de TIMER.
#include "C:\pic18\ biblioteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.

```

```

// Fosc = 20MHz
// Tciclo = 4/Fosc = 0,2us
#pragma config FOSC = HS
#pragma config CPUDIV = OSC1_PLL2

#pragma config WDT = OFF           //Desabilita o Watchdog Timer (WDT).
#pragma config PWRT = ON           //Habilita o Power-up Timer (PWRT)
#pragma config BOR = ON            //Brown-out Reset (BOR) habilitado somente no hardware
#pragma config BORV = 1             //Voltagem do BOR é 4,33V.
#pragma config PBADEN = OFF        //RB0,1,2,3 e 4 configurado como I/O digital
#pragma config LVP = OFF           //Desabilita o Low Voltage Program.
#pragma config LPT1OSC = ON //TIMER 1 programado para operar no modo low-power.

```

```
unsigned char buffer[11];
```

```
unsigned char segundo, minuto, hora;
```

```
void ISR_alta_prioridade(void), //Protótipo da função de interrupção.
```

```
#pragma code int_alta=0x08 //Vetor de interrupção de alta prioridade.
void int_alta (void)
{
    _asm GOTO ISR_alta_prioridade _endasm //Desvia o programa para a função ISR_alta_prioridade.
}
#pragma code
```

```

#pragma interrupt ISR_alta_prioridade
void ISR_alta_prioridade(void)
{
    //Carrega o TMR1 com 32768, para gerar uma interrupção a cada 1 segundo.
    //Note que o TMR1L não é alterado, o que permite que ele continue incrementando.
    TMR1H = 0x80; //Carrega o valor 0x80 para dentro do registro TMR1H.
    PIR1bits.TMR1IF = 0; //Limpa o Flag bit.

    segundo++;

    if(segundo == 60)
    {
        minuto++;
        segundo = 0;

        if(minuto == 60)
        {
            hora++;
            minuto = 0;

            if(hora == 24)
                hora = 0;
        }
    }
}

void main (void)
{
    TRISA = 0b00011111; // RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11100111; // RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111; // RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0b00000000; // RD0 a RD7 – saída.
    TRISE = 0b00000000; // RE0 a RE2 – saída.

    OpenTimer1(TIMER_INT_OFF           //Desabilita a interrupção do TIMER 1.
              &T1_8BIT_RW          //Operação de leitura/escrita de 8bits, pois somente o TMR1H deve ser alterado no ISR.
              &T1_SOURCE_EXT       //Seleciona a fonte externa (T13CKI).
              &T1_OSC1EN_ON        //Habilita o oscilador.
              &T1_SYNC_EXT_OFF     //Sincronismo desabilitado. Modo assíncrono.
              &T1_PS_1_1);         //Prescaler igual a 1 (1:1).

    PORTD = 0x00; //Coloca a porta D em 0V.
    PORTE = 0x00; //Coloca a porta E em 0V.

    Delay10KTCYx (100);           //Gera um delay de 200 milissegundos. Tempo para estabilizar o oscilador.

    PIR1bits.TMR1IF = 0; //Limpa o Flag bit da interrupção de TIMER 1.
    IPR1bits.TMR1IP = 1; //Seleciona alta prioridade.
    PIE1bits.TMR1IE = 1; //Habilita a interrupção de TIMER 1.

    lcd_inicia( 0x28, 0xF, 0x06 ); //Inicializa o display LCD alfanumérico com quatro linhas de dados.
    lcd_LD_cursor (0); //Desliga o cursor.
    lcd_posicao (1,1); //Desloca o ponteiro para a L=1 e C=1.
    imprime_string_lcd("REAL TIME CLOCK"); //Coloca a mensgem no visor.

    RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo.
}

```

```
INTCONbits.GIEH = 1; //Habilita todas as interrupções de alta prioridade.
INTCONbits.GIEL = 1; //Habilita todas as interrupções de baixa prioridade.
```

```
segundo = 0; //Segundo atual.
minuto = 7; //Minuto atual
hora = 1; //Hora atual.
```

```
//Carrega o TMR1 com 32768, para gerar uma interrupção a cada 1 segundo
TMR1H = 0x80; //Carrega o valor 0x80 para dentro do registro TMR1H.
TMR1L = 0x00; //Zera o registro TMR1L.
```

```
while (1) //Looping infinito
{
    lcd_posicao(2,1); // Desloca o ponteiro para a L=2 e C=1.
    sprintf(buffer, "%02d%02d%02d", hora, minuto, segundo); //Envia a string formatada para o buffer
    imprime_buffer_lcd(buffer,8); //Coloca no visor os oito primeiros elementos do buffer
    Delay10KTCYx (100); //Gera um delay de 200 milissegundos. 100*10.000*0,2us= 200ms
}
}
```

```
*****
***** Projeto Capítulo 11 – TIMER2 *****
**
** Mantém o LED_1 aceso ou apagado durante aproximadamente um segundo.
**
** Fosc = 20MHz -> Ciclo de máquina = 4 Fosc = 0.2us
** Prescaler igual a 1 16 -> TMR2 é incrementado depois de 16 ciclos de máquina. #0,2us*16 = 3.2us.
** PR2 = 249 -> Quando o TMR2 for igual a 249 o postscaler será incrementado em 1. #3.2us*250 = 800us.
** Postscaler igual a 1:10 -> Quando o postscaler for igual a 10, uma interrupção será gerada #800us*10 = 8.000us
** quant_interrup == 125 -> Após 125 ocorrências de interrupção, o estado do LED_1 é invertido #8ms*125 = 1 seg.
**
** Autor: Alberto Noboru Miyadaira
*****
```

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <adc.h> //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <timers.h> //Adiciona a biblioteca de funções de TIMER.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)
```

```
void ISR_baixa_prioridade(void); //Protótipo da função de interrupção.
```

```
#define LED_1 PORTEbits.RE0 //Define outro nome para a estrutura.
```

```
unsigned char quant_interrup = 0; //Armazena a quantidade de interrupções.
```

```
#pragma code int_baixa=0x18 //Vetor de interrupção de baixa prioridade.
```

```
void int_baixa (void)
{
    _asm GOTO ISR_baixa_prioridade _endasm //Desvia o programa para a função ISR_baixa_prioridade.
}
```

```
#pragma code
```

```
#pragma interrupt ISR_baixa_prioridade
void ISR_baixa_prioridade(void)
{
    quant_interrup++;
```

```

if( quant_interrup == 125)
{
    LED_1 = ~LED_1; //Inverte o estado do LED_1.
    quant_interrup = 0; //Reinicia a contagem de interrupção.
}
PIR1bits.TMR2IF = 0; //Limpa o Flag bit.
}

void main (void)
{
TRISA = 0b00011111; // RA0 a RA4 – entrada e RA5 a RA6 – saída.
TRISB = 0b11100111; // RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
TRISC = 0b10111111; // RC0 a RC5 e RC7 – entrada e RC6 – saída.
TRISD = 0b00000000; // RD0 a RD7 – saída.
TRISE = 0b00000000; // RE0 a RE2 – saída.

PR2 = 249; //Registro de período.

OpenTimer2(TIMER_INT_OFF      //Desabilita a interrupção do TIMER 2.
          &T2_POST_1_10   //Postscaler igual a 10 (1:10);
          &T2_PS_1_16);  //Prescaler igual a 16 (1:16).

//Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital. (Capítulo 13)
//Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h.
CloseADC ();                  //Requer a biblioteca adc.h.

PORTD = 0x00; //Coloca a porta D em 0V.
PORTE = 0x00; //Coloca a porta E em 0V.

PIR1bits.TMR2IF = 0;           //Limpa o Flag bit da interrupção de TIMER 2.
IPR1bits.TMR2IP = 0;           //Seleciona baixa prioridade.
PIE1bits.TMR2IE = 1;           //Habilita a interrupção de TIMER 2.

RCONbits.IPEN = 1;             //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo.
INTCONbits.GIEH = 1;            //Habilita todas as interrupções de alta prioridade.
INTCONbits.GIEL = 1;            //Habilita todas as interrupções de baixa prioridade.

WriteTimer2 (0); //Limpa o registro.
while (1); //Looping infinito.
}

```

12

Módulo CCP/ECCP

A maioria dos microcontroladores da família PIC18 possui o módulo CCP (Capture, Compare ou PWM), e em alguns casos, o ECCP (*Enhanced Capture, Compare ou PWM*), que no caso do PIC18F4550, nada mais é que um módulo CCP com funções adicionais para o PWM.

Esses módulos são capazes de operar como modo Capture, Compare ou PWM. Veja a seguir a descrição de cada um.

- **Capture:** mede o tempo entre dois eventos.
- **Compare:** dispara um determinado evento em um período de tempo predeterminado.
- **PWM (modulação por largura de pulso):** controla a tensão entregue a uma determinada carga, modificando a largura de pulso do sinal, dentro de um período de tempo prefixado.

12.1 Módulo CCP/ECCP do PIC18F4550

O microcontrolador PIC18F4550 possui dois módulos CCP, CCP1 e CCP2, o CCP1 está implementado como ECCP. Cada módulo possui um registro de 16bits, o qual pode operar como um registro do Capture, Compare ou PWM.

Ambos os módulos CCP/ECCP utilizam o TIMER 1, 2 ou 3 como fonte de *clock*. O TIMER 1 e/ou TIMER 3 são destinados aos modos Capture ou Compare, enquanto o TIMER 2 é destinado ao modo PWM. Veja no tópico "TIMER 3" do Capítulo 11, como configurar as fontes de *clock* (TIMER 1 e TIMER 3) para os módulos CCP/ECCP operando no modo Capture ou Compare.



Para que o Capture ou Compare funcione corretamente, a fonte de *clock* (TIMER 1 ou TIMER 3) deve ser configurada como temporizador ou contador síncrono.

12.1.1 Modo PWM

O módulo CCP, quando configurado para operar no modo PWM, é capaz de fornecer através do pino CCPx, um sinal PWM com resolução de 10bits.

No módulo CCP2, o pino CCP2 é multiplexado com o pino RB3/CCP2 ou RC1/CCP2, cuja seleção pode ser feita através do bit CCP2MX (`#pragma config CCP2MX`). Veja como configurá-lo no Capítulo 6.

Veja a seguir o diagrama de bloco simplificado do PWM.

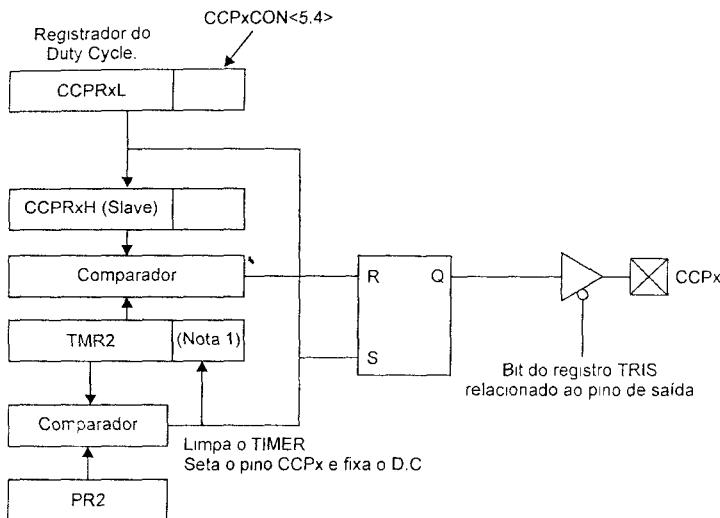


Figura 12.1: Diagrama de bloco simplificado do modo PWM. (PIC18F4550)

A modulação por largura de pulso (PWM) consiste em controlar o tempo em que o sinal permanecerá em nível alto (*duty cycle*), ou em outras palavras, controlar a largura de pulso, dentro de um período de tempo prefixado (Periodo_PWM), como pode ser verificado na Figura 12.2.

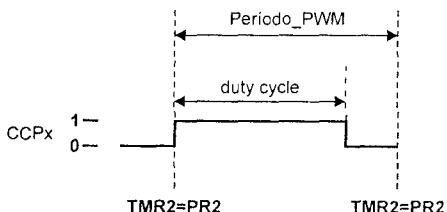


Figura 12.2: Saída de PWM no módulo CCPx.

Quando o modo PWM é selecionado, o módulo TIMER 2 é usado como fonte de *clock* para o módulo CCP/ECCP. Logo, o período de tempo do sinal (Periodo_PWM) é especificado pelo registro PR2 (o valor do registro é manipulado pela função `OpenPWM()`), e pode ser calculado de acordo com a equação apresentada em seguida:

$$\text{Periodo_PWM} = \frac{[(\text{PR2})+1] * 4 * (\text{valor_prescaler_TMR2})}{\text{Fosc}}$$

Se Fosc = 20MHz, temos que o período máximo de tempo para o sinal PWM é equivalente a:

$$\text{Período_PWM}_{\max} = \frac{[(255+1)*4*(16)]}{20M} = 819\mu\text{s}$$



Quando o valor do registro TMR2 é igual ao PR2, o TMR2 é reiniciado, o pino CCPx é colocado em nível alto (Vcc) e o valor do *duty cycle* (CCPRxL) é posto no registro CCPRxH.

O *duty cycle* (ciclo ativo) é especificado pelo registro **CCPRxL** (8bits mais significativos) e pelos *bits* **CCPxCON<5:4>** (2bits menos significativos).

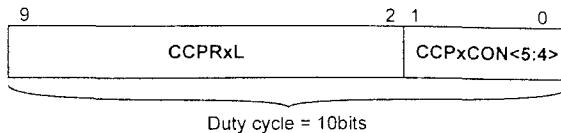


Figura 12.3: Registro de *duty cycle*.

O tempo de *duty cycle* pode ser obtido pela equação a seguir.

$$\text{Tempo_sinal_alto} = \frac{(\text{CCPRxL} \cdot \text{CCPxCON} < 5 : 4>) \cdot (\text{valor_prescaler_TMR2})}{\text{Fosc}}$$

Quando o valor de **CCPRxH:CCPxCON<5:4>** (10bits) for igual ao **TMR2** (8bits) concatenado com 2bits de *clock Q* ou 2bits do *prescaler* do TMR2, o pino CCPx será forçado para o nível baixo (V_{SS}), de acordo com a Figura 12.2.

A máxima resolução de PWM representada em *bits*, para uma dada frequência, é calculada pela equação a seguir.

$$\text{Resolução_PWM} = \frac{\log\left(\frac{\text{Fosc}}{\text{FPWM}}\right)}{\log(2)}$$

Sendo a frequência de PWM igual a 1/Período_PWM.

$$\text{FPWM} = \frac{1}{\text{Período_PWM}}$$

O módulo CCP1 também possui a função *auto-shutdown* presente no módulo ECCP, comentada mais adiante.



No modo PWM, os registros **CCPRxL:CCPxCON<5:4>** podem ser escritos em qualquer momento do programa, porém o valor do *duty cycle* será carregado para o registro **CCPRxH**, somente quando ocorrer a próxima igualdade dos registros **TMR2** e **PR2**. Neste modo, o registro **CCPRxH** é somente de leitura.

12.1.2 Módulo ECCP

O módulo ECCP comprehende todas as funcionalidades do módulo CCP padrão, além de possuir o PWM com capacidades aumentadas.

O microcontrolador PIC18F4550 possui apenas um módulo ECCP, o qual está implementado pelo módulo CCP1.

Praticamente toda explicação sobre o funcionamento do PWM até o presente momento é válida para o módulo ECCP, incluindo as equações de período, *duty cycle* e resolução de PWM. A principal diferença está no comportamento dos pinos de saída.

12.1.2.1 Modo PWM com Capacidade Aumentada

O PWM do módulo ECCP inclui:

- Múltiplas saídas (2 ou 4).
- Seleção de polaridade.
- Faixa inativa (*dead-band delay*) programável.
- Parada e reinício automático.

Veja na Figura 12.4 o diagrama de bloco do modo PWM do módulo ECCP.

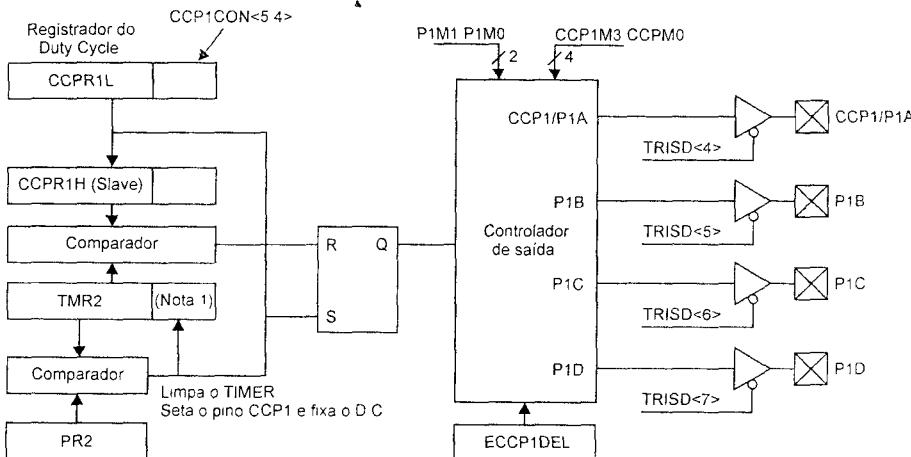


Figura 12.4: Diagrama de bloco do modo PWM do módulo ECCP. (PIC18F4550)

O PWM do módulo ECCP pode operar com uma simples saída (P1A), duas saídas (P1A e P1B - Meia ponte) ou quatro saídas (P1A, P1B, P1C e P1D - Ponte completa), cujos pinos devem ser configurados pelo registro **TRIS**.

O modo de operação do PWM com uma simples saída (P1A) tem um comportamento idêntico ao módulo CCP, e pode ser habilitado através da combinação dos bits P1M1:P1M0 (**CCP1CON<7:6>**) e CCP1M3:CCP1M0 (**CCP1CON<3:0>**).

Tabela 12.1: Configuração do PWM com uma simples saída (P1A).

P1M1:P1M0	CCP1M3:CCP1M0	Descrição
00	1100	Seleciona o modo PWM com uma única saída. O sinal modulado sai do pino P1A, ativo em nível alto (<i>active-high</i>). Observação: Os outros pinos (P1B, P1C e P1D) operam como pinos I/O
00	1101	Seleciona o modo PWM com uma única saída. O sinal modulado sai do pino P1A, ativo em nível alto (<i>active-high</i>). Observação: Os outros pinos (P1B, P1C e P1D) operam como pinos I/O.

P1M1:P1M0	CCP1M3:CCP1M0	Descrição
00	1110	Seleciona o modo PWM com uma única saída. O sinal modulado sai do pino P1A, ativo em nível baixo (<i>active-low</i>). Observação: Os outros pinos (P1B, P1C e P1D) operam como pinos I/O.
00	1111	Seleciona o modo PWM com uma única saída. O sinal modulado sai do pino P1A, ativo em nível baixo (<i>active-low</i>). Observação: Os outros pinos (P1B, P1C e P1D) operam como pinos I/O.

Veja a seguir os dois diferentes modos de saída de PWM sobre o pino P1A.

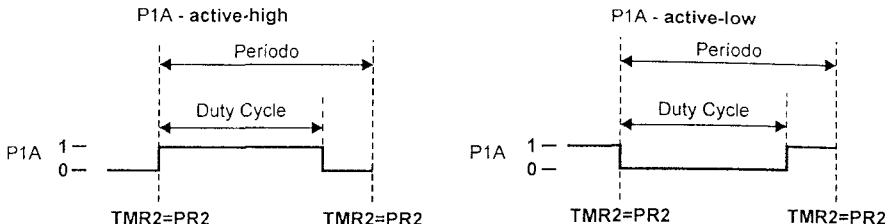


Figura 12.5: Saída de PWM no módulo CCP.

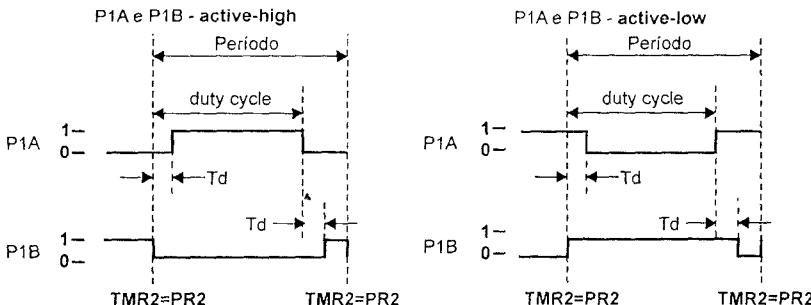
O segundo modo de operação do PWM é o *half-bridge* (meia ponte), podendo ser empregado em aplicações do tipo *half-bridge* e *full-bridge*. Uma vez habilitado, ambos os pinos P1A e P1B fornecem um sinal modulado, no entanto o controle do *duty cycle* é feito sobre o pino P1A e o seu complemento é posto no pino P1B. O modo de operação desses pinos pode ser configurado através dos bits P1M1:P1M0 (CCP1CON<7:6>) e CCP1M3:CCP1M0 (CCP1CON<3:0>).

Tabela 12.2: Configuração do PWM operando no modo *half-bridge*.

P1M1:P1M0	CCP1M3:CCP1M0	Descrição
10	1100	Seleciona o modo PWM operando em <i>half-bridge</i> . O sinal modulado sai do pino P1A e P1B. P1A ativo em nível alto (<i>active-high</i>) e P1B ativo em nível alto (<i>active-high</i>). Observação: Os outros pinos (P1C e P1D) operam como pinos I/O.
10	1101	Seleciona o modo PWM operando em <i>half-bridge</i> . O sinal modulado sai do pino P1A e P1B. P1A ativo em nível alto (<i>active-high</i>) e P1B ativo em nível baixo (<i>active-low</i>). Observação: Os outros pinos (P1C e P1D) operam como pinos I/O.
10	1110	Seleciona o modo PWM operando em <i>half-bridge</i> . O sinal modulado sai do pino P1A e P1B. P1A ativo em nível baixo (<i>active-low</i>) e P1B ativo em nível alto (<i>active-high</i>). Observação: Os outros pinos (P1C e P1D) operam como pinos I/O.
10	1111	Seleciona o modo PWM operando em <i>half-bridge</i> . O sinal modulado sai do pino P1A e P1B. P1A ativo em nível baixo (<i>active-low</i>) e P1B ativo em nível baixo (<i>active-low</i>). Observação: Os outros pinos (P1C e P1D) operam como pinos I/O.

Uma vez configurado, o *driver* passa a ser controlado pelo *hardware* do dispositivo, que dispõe de um mecanismo de segurança, o qual previne que a condução de corrente entre dois transistores resulte em um

curto-circuito. Isso é feito através da introdução de um atraso (*dead-band delay*) em cada transição dos pinos de controle (P1A e P1B), como pode ser observado na Figura 12.6.



Legenda: Td - Dead-Band Delay.

Figura 12.6: Estado dos pinos P1A e P1B no modo *half-bridge*.

Note que nesse modo (*half-bridge*), o *duty cycle* (Tempo_sinal_alto) pode não corresponder ao tempo calculado pelas equações apresentadas anteriormente, pois existe a possibilidade de programar um atraso entre as transições (Td). Esse atraso é decorrente do *dead-band delay* (Td), o qual é configurado através do registro **ECCP1DEL<6:0>** de 7bits, e corresponde ao elemento *dead_delay* da equação seguinte.

$$Td = 4 * \frac{1}{Fosc} * \text{dead_delay}$$

$$\text{dead_delay} = \frac{Td * Fosc}{4}$$

O registrador **ECCP1DEL** pode ser modificado com o auxílio da seguinte função:

```
void dead_delay (unsigned char valor_dead_delay)
{
    ECCP1DEL &= 0b10000000;
    ECCP1DEL = ECCP1DEL | ( valor_dead_delay & 0b01111111 );
}
```

Essa função previne que o oitavo bit do registrador **ECCP1DEL** não seja sobrescrito.

Veja a seguir o circuito eletrônico proposto pela Microchip para o modo *half-bridge*, operando na seguinte configuração: P1M1:P1M0 = 10 e CCP1M3:CCP1M0 = 1100.

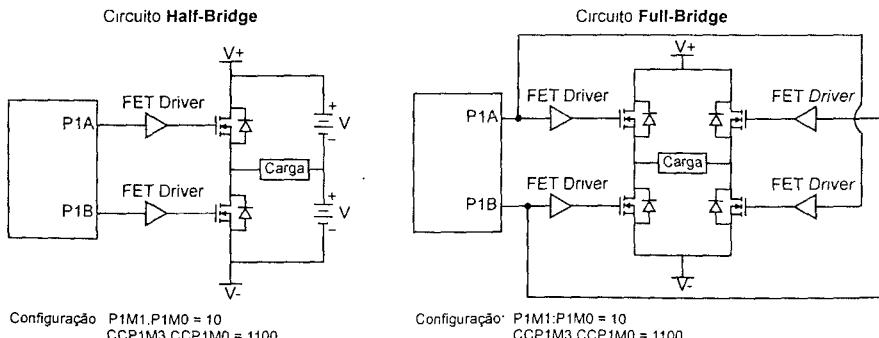


Figura 12.7: Circuito eletrônico proposto pela Microchip para o modo *half-bridge*.

O último modo possível de operação do PWM é o *full-brigde* que requer quatro pinos devidamente configurados como saída, denominados de P1A, P1B, P1C e P1D, cuja combinação, por exemplo, permite controlar a velocidade e direção de um motor DC.

A configuração desse modo depende da combinação dos bits P1M1:P1M0 (CCP1CON<7:6>) e CCP1M3:CCP1M0 (CCP1CON<3:0>).

Tabela 12.3: Configuração do PWM operando no modo *full-bridge*.

P1M1:P1M0	CCP1M3:CCP1M0	Descrição
01	1100	Seleciona o modo PWM operando em <i>full-bridge</i> . Direção: Sentido direto. O sinal modulado sai do pino P1D. (<i>active-high</i>). P1A é colocado em nível alto. P1B e P1C são colocados em nível baixo.
01	1101	Seleciona o modo PWM operando em <i>full-bridge</i> . Direção: Sentido direto. O sinal modulado sai do pino P1D. (<i>active-low</i>). P1C é colocado em nível baixo. P1A e P1B são colocados em nível alto.
01	1110	Seleciona o modo PWM operando em <i>full-bridge</i> . Direção: Sentido direto. O sinal modulado sai do pino P1D. (<i>active-high</i>). P1C é colocado em nível alto. P1A e P1B são colocados em nível baixo.
01	1111	Seleciona o modo PWM operando em <i>full-bridge</i> . Direção: Sentido direto. O sinal modulado sai do pino P1D. (<i>active-low</i>). P1A é colocado em nível baixo. P1B e P1C são colocados em nível alto.
11	1100	Seleciona o modo PWM operando em <i>full-bridge</i> . Direção: Sentido inverso. O sinal modulado sai do pino P1B. (<i>active-high</i>). P1C é colocado em nível alto. P1A e P1D são colocados em nível baixo.
11	1101	Seleciona o modo PWM operando em <i>full-bridge</i> . Direção: Sentido inverso. O sinal modulado sai do pino P1B. (<i>active-low</i>). P1A é colocado em nível baixo. P1C e P1D são colocados em nível alto.
11	1110	Seleciona o modo PWM operando em <i>full-bridge</i> . Direção: Sentido inverso. O sinal modulado sai do pino P1B. (<i>active-high</i>). P1A é colocado em nível alto. P1C e P1D são colocados em nível baixo.
11	1111	Seleciona o modo PWM operando em <i>full-bridge</i> . Direção: Sentido inverso. O sinal modulado sai do pino P1B. (<i>active-low</i>). P1C é colocado em nível baixo. P1A e P1D são colocados em nível alto.

Veja a seguir o estado dos pinos P1A, P1B, P1C e P1D no modo *full-bridge*.

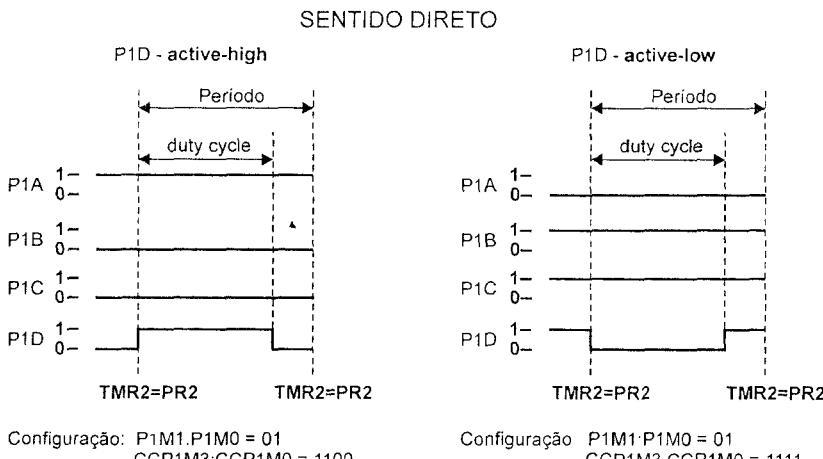


Figura 12.8: Estado dos pinos de saída no modo *full-bridge*. (Sentido direto)

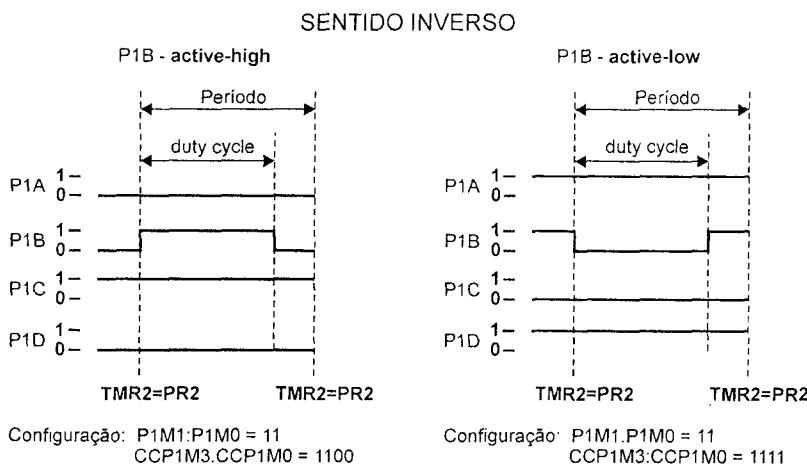


Figura 12.9: Estado dos pinos de saída no modo *full-bridge*. (Sentido inverso)

Observe que nesse modo somente dois pinos estão ativos ao mesmo tempo, e somente um pino (P1B ou P1D) é modulado. Além disso, não há o *dead-band delay*, resultando em um ciclo ativo (*duty cycle*) equivalente ao calculado. Veja a seguir o circuito proposto para o modo *full-bridge*, operando nas seguintes configurações.

Tabela 12.4: Bits de configuração do modo PWM para a ponte (*full-bridge*) representada na Figura 12.10.

P1M1:P1M0	CCP1M3:CCP1M0	Sentido
01	1100	Direto
11	1100	Inverso

Quando o módulo CCP está configurado como PWM, os pinos utilizados como saídas podem ser configurados para *auto-shutdown*. O *auto-shutdown* consiste em colocar o(s) pino(s) de saída do PWM em um estado de *shutdown* predefinido, quando um evento de *shutdown* for disparado.

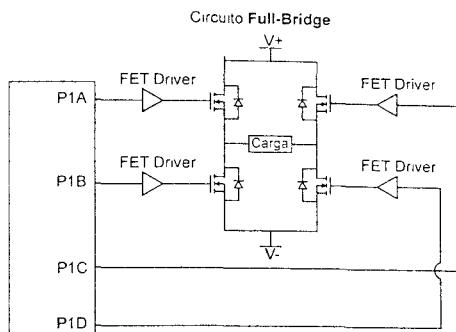


Figura 12.10: Circuito eletrônico proposto para o modo *full-bridge*.

Um evento de *shutdown* pode ser disparado nos seguintes casos:

- Saída do Módulo Comparador 1.
- Saída do Módulo Comparador 2.
- Verificação de nível baixo (V_{SS}) na entrada do pino FLT0.

Esse evento é configurado pelos bits ECCPAS2:ECCPAS0 (ECCP1AS<6:4>).

Tabela 12.5: Seleção do evento de *auto-shutdown*.

ECCPAS2:ECCPAS0	Descrição
111	FLT0 ou Comparador 1 ou Comparador 2
110	FLT0 ou Comparador 2.
101	FLT0 ou Comparador 1.
100	FLT0.
011	Ambos os Comparadores.
010	Saída do Comparador 2.
001	Saída do Comparador 1.
000	Desabilita o Auto-shutdown.



Os comparadores podem ser usados para monitorar a corrente consumida pela ponte. Caso ela exceda um determinado valor especificado pelo usuário, um *auto-shutdown* pode ser disparado.

O estado de *shutdown* dos pinos P1A e P1C de saída do PWM é definido pelos bits PSSAC1:PSSAC0 (ECCP1AS<3:2>), enquanto o estado dos pinos P1B e P1D é definido pelos bits PSSBD1:PSSBD0 (ECCP1AS<1:0>).

Tabela 12.6: Seleção do estado dos pinos P1A e P1C.

PSSAC1:PSSAC0	Descrição
1x	Os pinos P1A e P1C são colocados no modo <i>tri-state</i> .
01	Os pinos P1A e P1C são colocados em nível alto (V_{CC}).
00	Os pinos P1A e P1C são colocados em nível baixo (V_{SS}).

Tabela 12.7: Seleção do estado dos pinos P1B e P1D.

PSSBD1:PSSBD0	Descrição
1x	Os pinos P1B e P1D são colocados no modo <i>tri-state</i> .
01	Os pinos P1B e P1D são colocados em nível alto (V_{CC}).
00	Os pinos P1B e P1D são colocados em nível baixo (V_{SS}).

Se um evento de *auto-shutdown* for verificado pelo dispositivo, o *Flag bit ECCPASE (ECCP1AS<7>)* será setado. Se a opção *auto-shutdown* estiver habilitada, também pode reiniciar o módulo CCP de forma automática, após um evento de *shutdown*. Essa configuração é feita pelo *bit PRSEN (ECCP1DEL<7>)*.

- **PRSEN = 1:** o *Flag bit ECCPASE* é automaticamente limpo e o PWM é reiniciado.
- **PRSEN = 0:** o *Flag bit ECCPASE* deve ser limpo manualmente para reiniciar o PWM, enquanto isso os pinos permanecerão no estado de *shutdown*.

12.2 Funções para o Módulo CCP/ECCP

As funções de CCP/ECCP que serão apresentadas a seguir podem ser usadas em qualquer microcontrolador da família PIC18, e estão declaradas na biblioteca **capture.h** para o modo Capture, **compare.h** para o modo Compare e **pwm.h** para o modo PWM.



Se o modo de operação do PWM for *half-bridge*, será necessário adicionar a função **dead_delay ()** para configurar o *dead-band delay*.

Para o modo *full-bridge*, o recurso *auto-shutdown* é opcional, e caso queira utilizá-lo, é necessário configurá-lo manualmente.

12.2.1 Funções do Modo Capture

12.2.1.1 Desabilita o Capture

A função **CloseCapturex** desabilita o periférico Capture do módulo especificado, e a interrupção correspondente a ele. A função **CloseECapture1** só pode ser usada em dispositivos que possuam um registro **ECCPxCON**

Sintaxe

```
CloseCapture1()
CloseCapture2()
CloseCapture3()
CloseCapture4()
CloseCapture5()
CloseECapture1()
```

Exemplo

```
CloseCapture1(); //Desabilita o Capture do módulo CCP1.
CloseCapture2(); //Desabilita o Capture do módulo CCP2
```

12.2.1.2 Habilita o Capture

A função **OpenCapturex** habilita e configura o modo Capture. A função **OpenECapture1** só pode ser usada em dispositivos que possuam um registro **ECCPxCON**.

Sintaxe

```
OpenCapture1( configuração )
OpenCapture2( configuração )
OpenCapture3( configuração )
OpenCapture4( configuração )
OpenCapture5( configuração )
OpenECapture1( configuração )
```

Sendo:

- **configuração:** são as constantes listadas na Tabela 12.8, separadas por um '&'.

Tabela 12.8: Constantes de configuração do Capture.

Função	Constante	Descrição
Interrupção do módulo CCP	CAPTURE_INT_ON CAPTURE_INT_OFF	Habilita. Desabilita.
Disparo de interrupção	CAP_EVERY_FALL_EDGE CAP_EVERY_RISE_EDGE CAP_EVERY_4_RISE_EDGE CAP_EVERY_16_RISE_EDGE	Interrupção em todas as bordas de descida. Interrupção em todas as bordas de subida. Interrupção na 4 ^a borda de subida. Interrupção na 16 ^a borda de subida.
Observação: As constantes iniciadas com ECAP são destinadas ao modo <i>enhanced</i> Capture.	ECAP_EVERY_FALL_EDGE ECAP_EVERY_RISE_EDGE ECAP_EVERY_4_RISE_EDGE ECAP_EVERY_16_RISE_EDGE	Interrupção em todas as bordas de descida. Interrupção em todas as bordas de subida. Interrupção na 4 ^a borda de subida. Interrupção na 16 ^a borda de subida.

A biblioteca **capture.h** também possui uma estrutura chamada **CapStatus**, que indica o estado de *overflow* (estouro) de cada módulo Capture, cujos campos são:

- | | |
|-----------|------------|
| ▪ Cap1OVF | ▪ Cap4OVF |
| ▪ Cap2OVF | ▪ Cap5OVF |
| ▪ Cap3OVF | ▪ ECap1OVF |



O TIMER associado ao módulo deve ser configurado antes de habilitar o modo Capture.

Exemplo

```
OpenCapture1();  
OpenCapture2();
```

12.2.1.3 Operação de Leitura do Capture

A função **ReadCapturex** retorna o valor do registro do módulo CCP/ECCP usado pelo modo Capture. A função **ReadECapture1** só pode ser usada em dispositivos que possuam um registro **ECCPxCON**.

Sintaxe

```
valor = ReadCapture1()  
valor = ReadCapture2()  
valor = ReadCapture3()  
valor = ReadCapture4()  
valor = ReadCapture5()  
valor = ReadECapture1()
```

Sendo:

- **valor:** valor de 16bits correspondente ao valor dos registros do módulo CCP/ECCP.

Exemplo

```
while(!PIR1bits.CCP1IF); //Aguarda a ocorrência de um evento de Capture do módulo CCP1.  
Tempo_16bits = ReadCapture1(); //Lê o registro do módulo CCP1.
```

12.2.2 Funções do Módulo Compare

12.2.2.1 Desabilita o Compare

A função **CloseComparex** desabilita o periférico Compare do módulo especificado, e a interrupção correspondente a ele.

Sintaxe

```
CloseCompare1()
CloseCompare2()
CloseCompare3()
CloseCompare4()
CloseCompare5()
CloseECompare1()
```

Exemplo

```
CloseCompare1(); //Desabilita o Compare do módulo CCP1.
```

12.2.2.2 Habilita o Compare

A função **OpenComparex** habilita e configura o modo Compare.

Sintaxe

```
OpenCompare1( configuração, período )
OpenCompare2( configuração, período )
OpenCompare3( configuração, período )
OpenCompare4( configuração, período )
OpenCompare5( configuração, período )
OpenECompare1( configuração, período )
```

Sendo:

- **configuração**: são as constantes listadas na Tabela 12.9, separadas por um '&'.
- **período**: valor de 16bits (1 - 65535) que será comparado com o **TMRx**.

Tabela 12.9: Constantes de configuração do Compare.

Função	Constante	Descrição
Interrupção do módulo CCP	COM_INT_ON COM_INT_OFF	Habilita. Desabilita.
Disparo de interrupção	COM_TOGG_MATCH COM_HI_MATCH COM_LO_MATCH COM_UNCHG_MATCH COM_TRIG_SEVNT ECOM_TOGG_MATCH ECOM_HI_MATCH ECOM_LO_MATCH ECOM_UNCHG_MATCH ECOM_TRIG_SEVNT	Inverte o sinal de saída no pino CCPx. Seta o pino CCPx para o nível alto (Vcc). Seta o pino CCPx para o nível baixo (Vss). O sinal de saída no pino CCPx não é alterado. Dispara um evento especial. Inverte o sinal de saída no pino CCPx. Seta o pino CCPx para o nível alto (Vcc). Seta o pino CCPx para o nível baixo (Vss). O sinal de saída no pino CCPx não é alterado. Dispara um evento especial.
Observação : As constantes iniciadas com ECOM são destinadas ao modo <i>enhanced</i> Compare.		



O TIMER associado ao módulo deve ser configurado antes de habilitar o modo Compare.

No microcontrolador PIC18F4550, o evento especial consiste em reiniciar o TIMER associado ao módulo CCPx e iniciar uma conversão A/D, desde que o módulo A/D esteja devidamente configurado e habilitado.

Exemplo

```
SetTmrCCPSrc( T3_SOURCE_CCP );
OpenCompare1( COM_INT_ON & COM_TRIG_SEVNT , 60000);
```

12.2.3 Funções para o Modo PWM

12.2.3.1 Desabilita o PWM

A função **ClosePWMx** desabilita o periférico PWM do módulo especificado, e a interrupção correspondente a ele. A função **CloseEPWM1** só pode ser usada em dispositivos que possuam um registro **ECCPxCON**.

Sintaxe

```
ClosePWM1()
ClosePWM2()
ClosePWM3()
ClosePWM4()
ClosePWM5()
CloseEPWM1()
```

Exemplo

```
ClosePWM1(); //Desabilita o PWM do módulo CCP1
ClosePWM2(); //Desabilita o PWM do módulo CCP2.
```

12.2.3.2 Habilita o PWM

A função **OpenPWMx** determina o período do sinal PWM. Quando executada, ela carrega o valor do argumento *período* para dentro do registro **PR2** do TIMER especificado. A função **OpenEPWM1** só pode ser usada em dispositivos que possuam um registro **ECCPxCON**.



Essas funções usam somente o TIMER 2, o qual deve ser configurado antes da chamada destas.

Sintaxe

```
OpenPWM1( período )
OpenPWM2( período )
OpenPWM3( período )
OpenPWM4( período )
OpenPWM5( período )
OpenEPWM1( período )
```

Sendo:

- **período**: valor de 8bits (0 a 255). Esse argumento determina o período do PWM de acordo com a seguinte equação.

$$\text{Período_PWM} = \frac{[(\text{período}) + 1] * 4 * (\text{valor_prescaler_TMR2})}{\text{Fosc}}$$

12.2.3.3 Seta o Duty Cycle do Sinal PWM

A função **SetDCPWMx** escreve um novo valor de *duty cycle* no registro do PWM. A função **SetDCEPWM1** só pode ser usada em dispositivos que possuam um registro **ECCPxCON**.

Sintaxe

```
SetDCPWM1( duty_cycle )
SetDCPWM2( duty_cycle )
SetDCPWM3( duty_cycle )
SetDCPWM4( duty_cycle )
SetDCPWM5( duty_cycle )
SetDCEPWM1( duty_cycle )
```

Sendo:

- **duty_cycle**: valor de 10bits.

O tempo ativo do sinal PWM pode ser obtido pela seguinte equação.

$$\text{Tempo_sinal_ativo} = \frac{(\text{duty_cycle}) * (\text{valor_prescaler_TMR2})}{\text{Fosc}}$$

O valor máximo do argumento *duty_cycle* está diretamente ligado ao período do PWM. Logo, temos que:

$$\text{duty_cycle}_{\text{máx}} = \frac{(\text{Período_PWM}) * (\text{Fosc})}{\text{valor_prescaler_TMR2}}$$

Suponha que o sinal de *clock* do sistema seja 20MHz (Fosc = 20MHz), PR2 = 199 e o TIMER 2 esteja configurado com *prescaler* = 16. Logo, temos o seguinte período de tempo:

$$\text{Período_PWM} = \frac{[(199) + 1] * 4 * (16)}{20M} = 640\mu\text{s}$$

Portanto, o valor máximo do argumento *duty_cycle* é 800.

$$\text{duty_cycle}_{\text{máx}} = \frac{(640\mu\text{s}) * (20M)}{16} = 800$$

Exemplo

```
#include <sp18f2550.h>           //Arquivo de cabeçalho do PIC18F2550.
#include <timers.h>                //Adiciona a biblioteca de funções de TIMER.
#include <pwm.h>                  //Adiciona a biblioteca de funções de PWM.
```

```
// Fosc = 10MHz
// Tciclo = 4.Fosc = 0.4us
#pragma config FOSC = HS
```

```
#pragma config WDT = OFF //Desabilita o Watchdog Timer (WDT).
#pragma config LVP = OFF //Desabilita o Low Voltage Program.
```

```
void main (void)
```

```
{
unsigned int taxa_pwm = 0; //Largura do pulso do PWM.

TRISC = 0b11100000; //RC0 a RC4 – saída e RC5 a RC7 – entrada.

OpenTimer2(TIMER_INT_OFF //Desabilita a interrupção do TIMER 2.
           &T2_PS_1_4); //Prescaler igual a 4 (1:4)

OpenPWM1 (149); //PR1 = 149 -> Período_PWM = 240us. Logo, o máximo valor da variável taxa_pwm será 600.
OpenPWM2 (239); //PR2 = 239 -> Período_PWM = 384us. Logo, o máximo valor da variável taxa_pwm será 960.

SetDCPWM1(60); //Tsinal_alto = 60*4/10M = 24us.
SetDCPWM2(120); //Tsinal_alto = 120*4/10M = 48us.

while (1); //Looping infinito.
}
```

12.2.3.4 Define a Saída de PWM do Módulo ECCP

A função **SetOutputPWMx** configura a saída de PWM do módulo ECCP. A função **SetOutputEPWM1** só pode ser usada em dispositivos que possuam um registro **ECCPxCON**.

Sintaxe

```
SetOutputPWM1 ( config_saida, modo_saida )
SetOutputPWM2 ( config_saida, modo_saida )
SetOutputPWM3 ( config_saida, modo_saida )
SetOutputEPWM1 ( config_saida, modo_saida )
```

Sendo:

- **config_saida**: constante listada na Tabela 12.10.
- **modo_saida**: constante listada na Tabela 12.11.

Tabela 12.10: Constantes relacionadas à configuração do modo de operação do módulo PWM.

Constante	Descrição	PIC18F4550 (P1M1:P1M0)
SINGLE_OUT	Somente uma saída.	00
FULL_OUT_FWD	Saída do tipo <i>full-bridge</i> , com sentido direto.	01
HALF_OUT	Saída do tipo <i>half-bridge</i> .	10
FULL_OUT_REV	Saída do tipo <i>full-bridge</i> , com sentido inverso.	11

Tabela 12.11: Constantes relacionadas à configuração do modo de operação do módulo PWM.

Constante	Descrição	PIC18F4550 (CCP1M3:CCP1M0)
PWM_MODE_1	P1A e P1C <i>active-high</i> . P1B e P1D <i>active-high</i> .	1100
PWM_MODE_2	P1A e P1C <i>active-high</i> . P1B e P1D <i>active-low</i> .	1101
PWM_MODE_3	P1A e P1C <i>active-low</i> . P1B e P1D <i>active-high</i> .	1110
PWM_MODE_4	P1A e P1C <i>active-low</i> . P1B e P1D <i>active-low</i> .	1111

12.3 Projetos

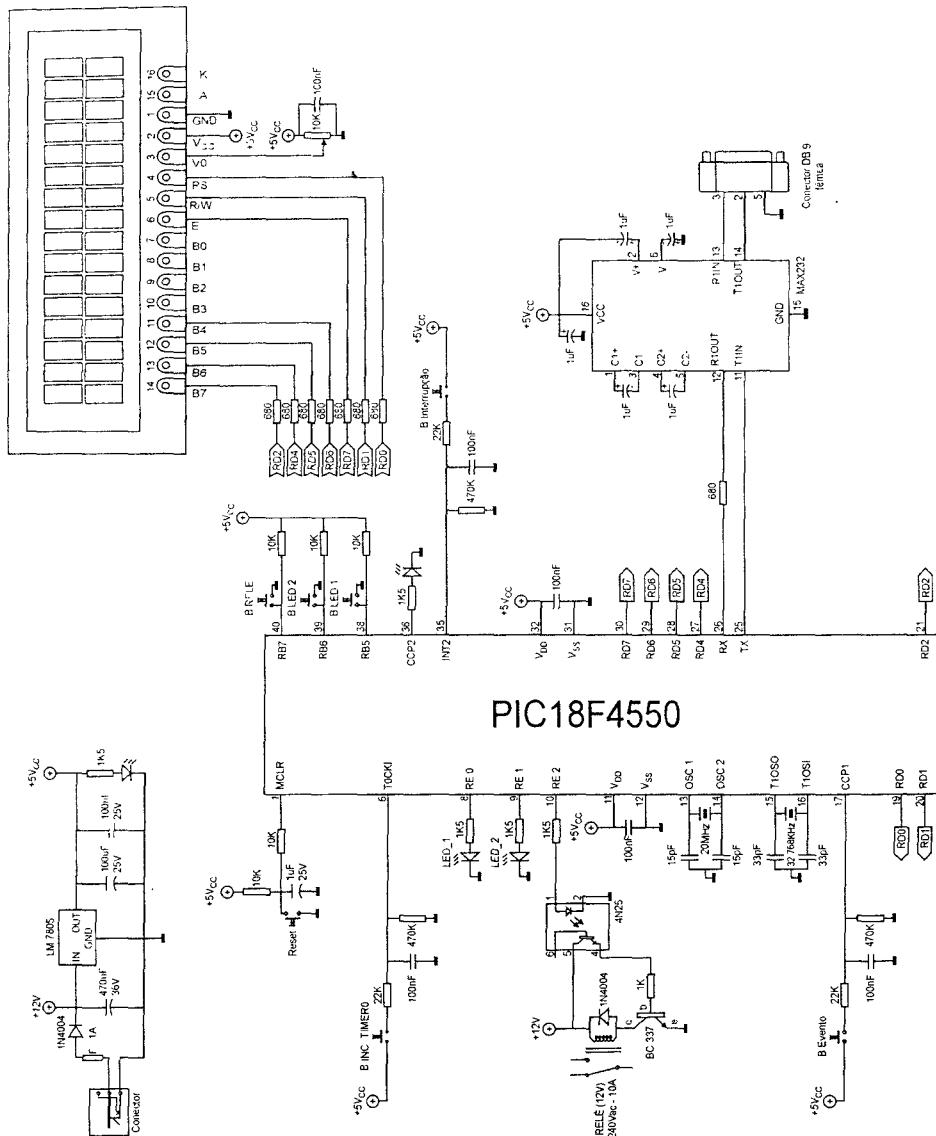


Figura 12.11: Circuito eletrônico para teste do módulo CCP.

12.3.1 Capture

Código

```
/*
*****Projeto Capítulo 12 - CAPTURE*****
*/
** Este programa permite calcular o tempo transcorrido entre dois eventos.
** Quando o botão B EVENTO é pressionado, o contador do TIMER1 é zerado e a contagem iniciada.
** A contagem é interrompida quando o botão B EVENTO é liberado.
** O tempo transcorrido é mostrado no display LCD 2x16.
**
** Autor: Alberto Noboru Miyadaira
*****/
```

```
#include <p18f4550.h>           //Arquivo de cabeçalho do PIC18F4550.
#include <stdio.h>              //Adiciona a biblioteca padrão de entrada e saída.
#include <stdlib.h>             //Adiciona a biblioteca de funções miscelâneas.
#include <timers.h>             //Adiciona a biblioteca de funções de TIMER.
#include <string.h>              //Adiciona a biblioteca de funções de manipulação de string.
#include <capture.h>             //Adiciona a biblioteca de funções de Capture.
#include "C:\pic18\ biblioteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)
```

```
void ISR_alta_prioridade(void); //Protótipo da função de interrupção.
void ISR_baixa_prioridade(void); //Protótipo da função de interrupção.
```

```
//Declaração das variáveis globais.
unsigned long quant_interrup = 0; //Armazena o número de estouros do contador do TIMER1.
unsigned long tempo_evento = 0;   //Registra o tempo transcorrido entre dois eventos.
unsigned int tempo_fim = 0;       //Armazena o valor dos registradores CCPR1H e CCPR1L.
unsigned char buffer[16];
unsigned char var_32[11];
```

```
//Informa a ocorrência da interrupção CCP1. 0 - B EVENTO pressionado e 1 - B EVENTO liberado.
unsigned char ocorrencia = 0;
```

```
#pragma code int_baixa=0x18 //Vetor de interrupção de baixa prioridade.
void int_baixa (void)
{
    _asm GOTO ISR_baixa_prioridade _endasm //Desvia o programa para a função ISR_baixa_prioridade.
}
#pragma code
```

```
/*
A interrupção do TIMER1 é gerada sempre que ocorrer um estouro do contador do TIMER1.
Valor máximo do contador = 65535.
Esta função regista na variável quant_interrup, a quantidade de vezes que ocorreu o estouro.
*/
#pragma interrupt ISR_baixa_prioridade
void ISR_baixa_prioridade(void)
{
    quant_interrup++;
    PIR1bits.TMR1IF = 0; //Limpa o Flag bit da interrupção de TIMER 1.
}
*/
No primeiro momento, a interrupção CCP1 é gerada quando o botão B EVENTO é pressionado. (borda de subida)
```

Quando o botão é pressionado, as seguintes ações são realizadas:

- 1 - O contador do TIMER1 é zerado.
- 2 - O contador de interrupções do TIMER1 (quant_interrup) é zerado.

- 3 - A variável ocorrência é setada para 1. (Esta variável informa que o botão está pressionado.)
 4 - O sentido da borda para gerar a interrupção é invertido.
 Desta forma, a interrupção ocorre novamente, quando o botão B EVENTO for liberado (borda de descida)

Quando o botão é liberado, a função desabilita as interrupções e calcula o tempo transcorrido entre os eventos.
 Adotando os seguintes procedimentos:

1 - Registra o valor dos registradores CCPR1H e CCPR1L na variável tempo_fim.

2 - Calcula a quantidade total que o contador do TIMER1 realizou.

Equação: tempo_evento = 65536*quant_interrup + tempo_fim.

3 - Transforma o valor contado em um valor correspondente ao tempo transcorrido [ms].

Equação: tempo_evento = tempo_evento*Tciclo_de_máquina. [ms]

Tciclo_de_máquina = 4/Fosc [us] = 4/(Fosc/1000) [ms] = 4/(20000000/1000) = 4/20000. [ms]

Logo.

Equação: tempo_evento = (tempo_evento*4)/20000. [ms]

4 - Mostra o tempo no display LCD 2x16.

5 - A variável ocorrência é setada para 0. (Esta variável informa que o botão está liberado.)

6 - O sentido da borda para gerar a interrupção é invertido.

Desta forma, a interrupção ocorre, quando o botão B EVENTO for pressionado. (borda de subida)

7 - As interrupções são habilitadas novamente.

*/

```
#pragma code int_alta=0x08 //Vetor de interrupção de alta prioridade, ou padrão.
void int_alta (void)
{
  _asm GOTO ISR_alta_prioridade _endasm //Desvia o programa para a função ISR_alta_prioridade.
}
#pragma code

#pragma interrupt ISR_alta_prioridade
void ISR_alta_prioridade(void)
{
  if (ocorrencia == 0)//Se o botão B EVENTO for pressionado, a rotina dentro do if será executada.
  {
    quant_interrup = 0;
    ocorrencia = 1; //Informa que o botão está pressionado.
    WriteTimer1(0); //Zera o registro do TIMER 1.
    OpenCapture1(CAPTURE_INT_ON //Habilita a interrupção do módulo CCP1.
                 &CAP_EVERY_FALL_EDGE); //Captura na borda de descida.
  }
  else //Se o botão B EVENTO for solto, então é executada a rotina dentro do else.
  {
    RCONbits.IPEN = 0;//Desabilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo.

    //Variável tempo_fim recebe o valor dos registradores do módulo CCP1.
    tempo_fim = ReadCapture1();

    //Cálculo do tempo entre os eventos.
    tempo_evento = 65536*quant_interrup + tempo_fim;
    tempo_evento = tempo_evento*4/20000; //Retorna o valor em microsegundos.
    ultoa (tempo_evento,var_32); //Converte o valor unsigned long em uma string.

    lcd_limpia_tela( );//Limpa a tela do display LCD e desloca o ponteiro para a L=1 e C=1
    imprime_string_lcd ("Tempo eventos. "); //Imprime uma string no display.
    lcd_posicao (2,1); // Desloca o ponteiro para a L=2 e C=1.
    sprintf(buffer, "T: %sms"\var_32); //Envia a string formatada para o buffer.
    imprime_buffer_lcd(buffer,strlen(var_32)+5); //Coloca no visor, os 14 primeiros elementos do buffer.

    ocorrencia = 0; //Informa que o botão não está sendo pressionado.
  }
}
```

```

    OpenCapture1(CAPTURE_INT_ON           //Habilita a interrupção do módulo CCP1.
    &CAP_EVERY_RISE_EDGE); //Captura na borda de subida

    RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo
}

void main (void)
{
    TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0b00000000; //RD0 a RD7 – saída
    TRISE = 0b00000000; //RE0 a RE2 – saída.

    OpenTimer1(TIMER_INT_ON           //Habilita a interrupção do TIMER 1.
    &T1_16BIT_RW               //Operação de 16bits.
    &T1_SOURCE_INT             //Seleciona a fonte de clock interna (Fosc/4)
    &T1_OSC1EN_OFF            //Desabilita o oscilador.
    &T1_PS_1_1);              //Prescaler igual a 1 (1:1).
SetTmrCCPSrc(T1_SOURCE_CCP); //TIMER 1 para o CCP1 e CCP2
IPR1bits.TMR1IP = 0; //Seta a interrupção de TIMER 1 como baixa prioridade.

OpenCapture1(CAPTURE_INT_ON           //Habilita a interrupção do módulo CCP1.
    &CAP_EVERY_RISE_EDGE); //Captura na borda de subida.

IPR1bits CCP1IP = 1; //Seta a interrupção de Capture 1 como alta prioridade.

PORTD = 0x00; //Coloca a porta D em 0V
PORTE = 0x00; //Coloca a porta E em 0V.

lcd_inicia(0x28, 0xF, 0x06); //Iniciaiza o display LCD alfanumérico com quatro linhas de dados
lcd_LD_cursor(0); //Desliga o cursor.
lcd_posicao(1,1); //Desloca o ponteiro para a L=1 e C=1.
imprime_string_lcd ("Tempo eventos "); //Imprime uma string no display.

RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo.
INTCONbits.GIEH = 1; //Habilita todas as interrupções de alta prioridade.
INTCONbits.GIEL = 1; //Habilita todas as interrupções de baixa prioridade.

while (1); //Looping infinito.
}

```

12.3.2 Compare

Código

```

***** Projeto Capítulo 12 - COMPARE ****
*
** Este programa permite ligar e desligar o LED _1 em um período de tempo definido.
** Uma interrupção é gerada, quando o valor do TIMER3 for igual ao valor da variável valor_comparado.
**
** Autor: Alberto Noboru Miyadaira
*****
```

```

#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <timers.h> //Adiciona a biblioteca de funções de TIMER.
#include <adc.h> //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <compare.h> //Adiciona a biblioteca de funções de Compare.
#include "C:\pic18\ biblioteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.
```

```
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)
```

```
void ISR_alta_prioridade(void); //Protótipo da função de interrupção.
```

```
#define LED_1 PORTEbits.RE0 //Define outro nome para a estrutura
```

```
#pragma code int_alta=0x08 //Vetor de interrupção de alta prioridade, ou padrão.
```

```
void int_alta (void)
```

```
{
    _asm GOTO ISR_alta_prioridade _endasm //Desvia o programa para a função ISR_alta_prioridade.
```

```
}
```

```
#pragma code
```

```
#pragma interrupt ISR_alta_prioridade
```

```
void ISR_alta_prioridade(void)
```

```
{
```

```
    LED_1 =~ LED_1; //Inverte o status do LED_1.
```

```
    PIR1bits.CCP1IF = 0; //Limpa o Flag bit da interrupção.
```

```
}
```

```
void main (void)
```

```
{
```

```
    unsigned int valor_comparado; //Valor que será comparado com o TIMER1
```

```
TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
```

```
TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída
```

```
TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
```

```
TRISD = 0b00000000; //RD0 a RD7 – saída.
```

```
TRISE = 0b00000000; //RE0 a RE2 – saída.
```

```
OpenTimer3(TIMER_INT_OFF //Desabilita a interrupção do TIMER 1.
```

```
&T3_16BIT_RW //Operação de 16bits.
```

```
&T3_SOURCE_INT //Seleciona a fonte de clock interna (Fosc/4).
```

```
&T3_PS_1_8); //Prescaler igual a 8 (1:8).
```

```
//Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital. (Capítulo 13)
```

```
//Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
```

```
OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h.
```

```
CloseADC ( ); //Requer a biblioteca adc.h
```

```
//Configuração do modo Compare do módulo CCP1
```

```
//Se ocorrer uma igualdade dos registros TMR3 e CCPR
```

```
//Uma interrupção será gerada e o TIMER 3 será reiniciado.
```

```
//O pino CCP1 não é afetado.
```

```
SetTmrCCPSrc ( T3_SOURCE_CCP );
```

```
OpenCompare1( COM_INT_ON & COM_TRIG_SEVNT , 60000);
```

```
PIR1bits.CCP1IF = 0; //Limpa o Flag bit do módulo CCP1.
```

```
PIR1bits.CCP1IP = 1; //Seta a interrupção de Capture 1 como alta prioridade.
```

```
PIE1bits.CCP1IE = 1; //Habilita a interrupção do módulo CCP1.
```

```
PORTD = 0x00; //Coloca a porta D em 0V.
```

```
PORTE = 0x00; //Coloca a porta E em 0V.
```

```
Lcd_inicia(0x28, 0xF, 0x06); //Inicializa o display LCD alfanumérico com quatro linhas de dados.
```

```
Lcd_LD_cursor (0); //Desliga o cursor.
```

```
Lcd_posicao (1,1); // Desloca o ponteiro para a L=1 e C=1.
```

```
imprime_string_lcd ("Teste modo COMPARE "); //Imprime uma string no display.
```

RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade Endereço 0x08 - alto e 0x18 - baixo
INTCONbits.GIEH = 1; //Habilita todas as interrupções de alta prioridade.
INTCONbits.GIEL = 1; //Habilita todas as interrupções de baixa prioridade.

```
while (1); //Looping infinito
}
```

12.3.3 PWM

Código

```
*****Projeto Capítulo 12 – PWM*****
**
** Este programa permite controlar o brilho do LED, através do PWM2.
**
** Autor: Alberto Noboru Miyadaira
*****/
```

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <timers.h> //Adiciona a biblioteca de funções de TIMER
#include <adc.h> //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída.
#include <pwm.h> //Adiciona a biblioteca de funções de PWM.
#include "C:\pic18\ biblioteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.
```

```
// Fosc = 20MHz
// Tciclo = 4/Fosc = 0.2us
#pragma config FOSC = HS
#pragma config CPUDIV = OSC1_PLL2

#pragma config WDT = OFF //Desabilita o Watchdog Timer (WDT)
#pragma config PWRT = ON //Habilita o Power-up Timer (PWRT).
#pragma config BOR = ON //Brown-out Reset (BOR) habilitado somente no hardware.
#pragma config BORV = 1 //Voltagem do BOR é 4,33V.
#pragma config PBADEN = OFF //RB0,1,2,3 e 4 configurado como I/O digital.
#pragma config LVP = OFF //Desabilita o Low Voltage Program.
#pragma config CCP2MX = OFF //Sinal de PWM no pino RB3.
```

```
//Variáveis globais
unsigned int validade_tecla, cont_tecla;
unsigned char tecla, tecla_anterior;
unsigned char buffer[16];
```

```
#define B_DIMINUI PORTBbits.RB5 //Define outro nome para a estrutura.
#define B_AUMENTA PORTBbits.RB6 //Define outro nome para a estrutura.
#define B_CONTINUO PORTBbits.RB7 //Define outro nome para a estrutura.
```

```
//Retorna o valor(0 ou 1) invertido do status do botão informado.
unsigned char teclado (unsigned char valor)
{
    switch (valor)
    {
        case 1: return B_AUMENTA; //Retorna o valor invertido do status do B_LED1.
        case 2: return B_DIMINUI; //Retorna o valor invertido do status do B_LED2
    }
}
```

```
//Filtro implementado em software, verifica 3.000 amostras.
unsigned char filtro_teclado (unsigned char num_tecla)
```

```

{
    validade_tecla=0;

    for( cont_tecla = 0; cont_tecla < 3000 ; cont_tecla++ )
    {
        validade_tecla = validade_tecla + teclado( num_tecla );
    }

    if(validade_tecla < 10)
        return num_tecla;
    else
        return 0;
}

//Atribui um valor ao botão, e faz uma verificação da validade dele.
unsigned char checa_tecla(unsigned char desc_tecla_ant)
{
    tecla=0;

    if (desc_tecla_ant == 0)//Desconsidera a tecla anterior.
        tecla_anterior = 0;

    if ( B_AUMENTA == 0 ) // Botão B_LED1 - Número do botão é 1.
        if( tecla_anterior != 1 )
            tecla = filtro_teclado(1);
        else
            tecla = 1;

    if ( B_DIMINUI == 0 )// Botão B_LED2 - Número do botão é 2.
        if(tecla_anterior!=2)
            tecla = filtro_teclado(2);
        else
            tecla = 2;

    if (tecla_anterior == tecla) //Se for a mesma tecla, então retorna 0.
        tecla = 0;
    else //Caso contrário, retorna o número correspondente à tecla.
        tecla_anterior = tecla;

    return tecla;
}

void main (void)
{
    unsigned int taxa_pwm = 0; //Largura do pulso do PWM.
    unsigned char botao_pres; //Número do botão.

    TRISA = 0b00011111;      //RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11100111;      //RB0,RB1,RB2,RB5,RB6 e RB7 - entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111;      //RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0b00000000;      //RD0 a RD7 – saída.
    TRISE = 0b00000000;      //RE0 a RE2 – saída.

    OpenTimer2(TIMER_INT_OFF          //Desabilita a interrupção do TIMER 2.
              &T2_PS_1_16);       //Prescaler igual a 16 (1:16)

    //Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital. (Capítulo 13)
    //Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
    OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h.
    CloseADC ();                  //Requer a biblioteca adc.h.

    PORTD = 0x00; //Coloca a porta D em 0V.
}

```

```
PORTE = 0x00; //Coloca a porta E em 0V.  
//PR2 = 199 -> Período_PWM = 640us. Logo, o máximo valor da variável taxa_pwm será 800  
OpenPWM2(199);  
SetDCPWM2(0); //Zera o duty cycle.  
  
lcd_inicia(0x28, 0x06); //Inicializa o display LCD aífanumérico com quatro linhas de dados  
lcd_LD_cursor(0); //Desliga o cursor.  
  
lcd_posicao(1,1); //Desloca o ponteiro para a L=1 e C=1.  
imprime_string_lcd("Teste modo PWM "); //Imprime uma string no display.  
  
while (1) //Looping infinito.  
{  
    //Se o botão B_RELÉ estiver pressionado, as teclas serão reconhecidas a cada iteração.  
    if ( B_CONTINÚO == 1 )  
        botao_pres = checa_tecla(1); //Reconhece a tecla apenas uma vez.  
    else  
        botao_pres = checa_tecla(0); //Reconhece a tecla a cada iteração.  
  
    if( botao_pres == 1 ) //Se o botão B_LED2 for pressionado, a rotina é executada.  
    {  
        if(taxa_pwm < 800)  
            SetDCPWM2(taxa_pwm += 2); //Aumenta a largura do pulso.  
    }  
    if( botao_pres == 2 ) //Se o botão B_LED1 for pressionado, a rotina é executada.  
    {  
        if(taxa_pwm > 0)  
            SetDCPWM2(taxa_pwm -= 2); //Diminui a largura do pulso.  
    }  
    lcd_posicao(2,1); //Desloca o ponteiro para a L=2 e C=1  
    sprintf(buffer, "Taxa 0-800: %04u", taxa_pwm), //Envia a string formatada para o buffer  
    imprime_buffer_lcd(buffer,16); //Coloca no visor os elementos do buffer  
}  
}
```

13

Conversor Analógico-Digital

Um conversor analógico/digital (A/D) converte uma grandeza analógica (temperatura, luminosidade, umidade etc.) em um valor digital (representação binária) proporcional a ela. Esse valor depende diretamente da resolução e da tensão de referência (V_{ref}) utilizada na conversão, podendo variar de 0 a $(2^{\text{resolução}} - 1)$ unidades. Veja na sequência um exemplo de conversor A/D.

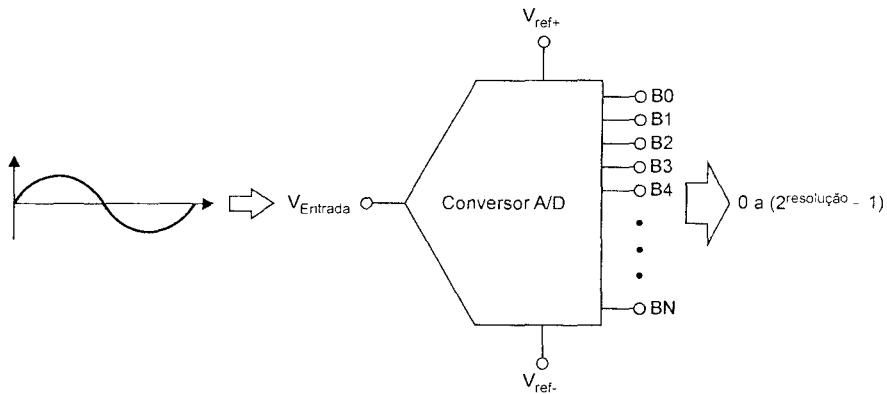


Figura 13.1: Conversor A/D.

O valor correspondente à variação de cada unidade pode ser obtido através da equação seguinte:

$$V_{\text{unidade}} = \frac{V_{\text{ref}+} - V_{\text{ref}-}}{2^{\text{resolução}} - 1}$$

O tempo de aquisição e a conversão também influenciam na conversão do sinal analógico, e serão comentados no decorrer deste capítulo.

13.1 Conversor A/D do PIC18F4550

O conversor A/D do microcontrolador PIC18F4550 possui 13 canais analógicos multiplexados (AN0 a AN12), uma resolução de 10 bits, com tensão de referência configurável, além de ser capaz de funcionar no modo IDLE ou SLEEP.

As conversões realizadas por este módulo são baseadas nas tensões V_{ref-} e V_{ref+} , e podem ser configuradas de quatro modos distintos

Tabela 13.1: Tensão de referência.

Tensão de referência para o canal analógico selecionado
$V_{ref-} = \text{Pino VREF- e } V_{ref+} = \text{Pino VREF+}$
$V_{ref-} = \text{Pino VREF- e } V_{ref+} = V_{CC}$
$V_{ref-} = V_{SS} \text{ e } V_{ref+} = \text{Pino VREF+}$
$V_{ref-} = V_{SS} \text{ e } V_{ref+} = V_{CC}$



Se as entradas analógicas AN8 a AN12 forem configuradas como canais de entrada do conversor A/D, a diretiva de configuração `#pragma config PBADEN = ON` deve ser inserida no programa.

Os pinos configurados como entrada analógica devem ser definidos como entrada através do registro TRIS.

13.1.1 Tempo de Aquisição e Conversão do Sinal

Sempre que um canal analógico é selecionado, ele deve ser carregado antes de a conversão ser iniciada. O tempo necessário para carregar esse canal é denominado de tempo de aquisição, e deve estar corretamente configurado, a fim de garantir que o capacitor C_{HOLD} do módulo A/D esteja completamente carregado com o nível de tensão aplicado no pino de entrada. Visto que, ao iniciar uma conversão A/D, o capacitor C_{HOLD} é desconectado do pino de entrada, e o valor de tensão presente nesse capacitor é convertido. A Figura 13.2 ilustra o modelo da entrada analógica do PIC18F4550.

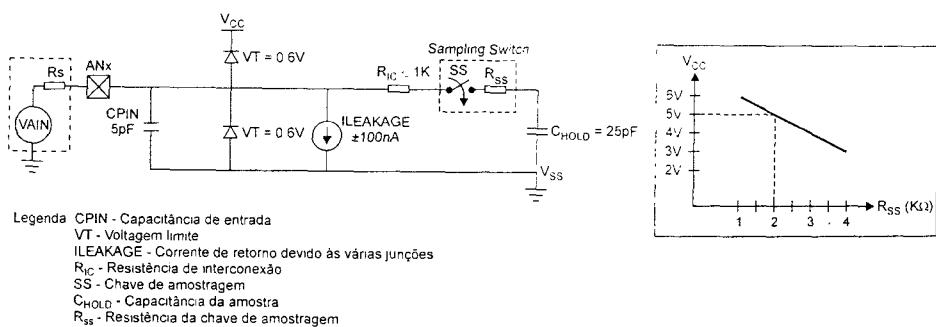


Figura 13.2: Modelo do canal analógico do PIC18F4550.

A equação que descreve o tempo mínimo de aquisição para dispositivos alimentados com 5V é:

$$T_{\min_aq_5V} = T_{AMP} + T_C + T_{COFF}$$

Sendo:

$$T_C = C_{HOLD} (R_{IC} + R_{SS} + R_S) \ln (1/2048)$$

$$T_{COFF} = [(T_{Amb} - 25^\circ\text{C})(0.02\mu\text{s}/^\circ\text{C})]$$

Sendo:

- T_{AMP} : tempo de estabilização do amplificador. ($T_{AMP} = 0.2\mu s$ para o PIC18F4550)
- T_C : tempo de carregamento do capacitor que mantém o valor do sinal. (C_{HOLD})
- T_{COFF} : coeficiente de temperatura.
- T_{Amb} : temperatura ambiente [°C].
- R_S : impedância de entrada. (Recomenda-se que $R_S \leq 2.5K\Omega$)
- R_{SS} : resistência da chave de amostragem (*Sampling Switch*). ($R_{SS} = 2K$)
- R_{IC} : resistência de interconexão. ($R_{IC} \leq 1K$)

Levando em consideração que $T_{AMP} = 0.2\mu s$ para o PIC18F4550. Temos que, o tempo mínimo de aquisição pode ser representado pela equação seguinte.

$$T_{min_aq_5V} = 0,7718 * us + 0,0001906 * R_S * us/ohm + (T_{amb} - 25^{\circ}C) * (0,02us/{}^{\circ}C)$$

Suponha que a resistência de entrada seja igual a $2.5K\Omega$ ($R_S = 2.5K\Omega$) e a temperatura ambiente igual a $50^{\circ}C$ ($T_{amb}=50^{\circ}C$), logo o tempo mínimo de aquisição para estas condições é igual a:

$$T_{min_aq_5V} = 0,7718 * us + 0,0001906 * 2,5K * us/ohm + (50^{\circ}C - 25^{\circ}C) * (0,02us/{}^{\circ}C) = 1,75us$$

O tempo de aquisição do módulo A/D pode ser configurado para ser gerado de modo manual ou automático.

Se o tempo de aquisição manual for selecionado quando um comando de conversão é executado, a aquisição é interrompida e a conversão iniciada. Portanto, o tempo de aquisição requerido entre a seleção da entrada e o comando de conversão deve ser assegurado pelo usuário. Por outro lado, se o tempo de aquisição automático estiver habilitado, o módulo A/D continuará amostrando a entrada por um período de tempo predeterminado, e automaticamente iniciará a conversão.

A principal vantagem do tempo de aquisição automático, é que não há necessidade de se preocupar com o tempo de aquisição entre a seleção do canal e o comando de conversão. Esse tempo pode ser definido como $20 T_{AD}$, $16 T_{AD}$, $12 T_{AD}$, $8 T_{AD}$, $6 T_{AD}$, $4 T_{AD}$ ou $2 T_{AD}$. Se for configurado como $0 T_{AD}$, é tratado como tempo de aquisição manual.

T_{AD} é o tempo de conversão de cada bit, devendo estar compreendido entre $0,7\mu s$ e $25\mu s$.

Tabela 13.2: Seleção da fonte de clock para a conversão A/D.

Descrição	Máximo Fosc	TAD (Mínimo)
FRC (Oscilador interno RC)	1.00MHz	TAD = $1/1MHz = 1\mu s$
Fosc/2	2.86MHz	TAD = $2/2.86MHz = 0,699\mu s$
Fosc/4	5.71MHz	TAD = $4/5.71MHz = 0,7\mu s$
Fosc/8	11.43MHz	TAD = $8/11.43MHz = 0,699\mu s$
Fosc/16	22.86MHz	TAD = $16/22.86MHz = 0,699\mu s$
Fosc/32	40.0MHz	TAD = $32/40MHz = 0,8\mu s$
Fosc/64	40.0MHz	TAD = $64/40MHz = 1,6\mu s$



Se a fonte de clock F_{RC} for selecionada, é adicionado um atraso equivalente a um ciclo de instrução ($Tcy = 4/Fosc$) antes de iniciar o clock de conversão A/D, permitindo que uma instrução Sleep () seja executada antes de iniciar uma conversão.

Suponha que $F_{osc} = 20MHz$, logo o menor valor válido de T_{AD} obtido na Tabela 13.2 é 0.8us obtido a partir da seguinte operação:

$$T_{AD} = \frac{1}{F_{osc}} = \frac{16}{20MHz} = 0.8\mu s$$

Agora que sabemos a definição de tempo de aquisição e tempo de conversão de cada bit (T_{AD}), veremos na sequência o modo de operação do módulo conversor A/D com/sem tempo de aquisição automático.

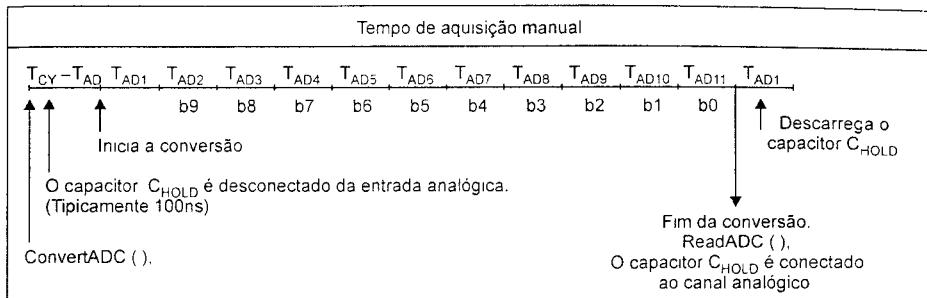


Figura 13.3: Ciclos T_{AD} de conversão A/D, sem tempo de aquisição automático.

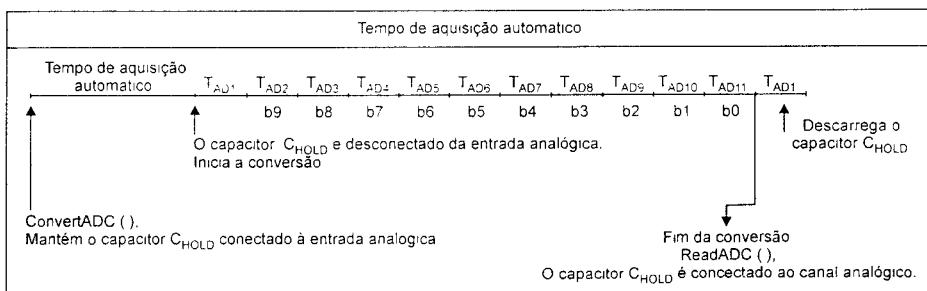


Figura 13.4: Ciclos T_{AD} de conversão A/D, com tempo de aquisição automático.

13.1.2 Conversão do Sinal Analógico

Vejamos a seguir, algumas equações envolvidas na conversão dos dados provenientes do conversor A/D. Suponha que as tensões de V_{ref-} e V_{ref+} do conversor sejam iguais a 0V e 5V respectivamente, logo temos que o valor de cada unidade ($V_{unidade}$) de saída do conversor corresponde a aproximadamente 4,887585mV.

$$V_{1bit} = \frac{V_{referência}}{2^{\text{resolução}} - 1} = \frac{5000mV}{1024 - 1} = \frac{5000mV}{1023} = 4,887585mV$$

Uma vez convertido, o valor do sinal analógico pode ser representado por uma variável interna pela equação seguinte.

$$\text{Valor}_{\text{conversor}} = \frac{V_{\text{sinal_analogico}} * (2^{\text{resolução}} - 1)}{V_{\text{referência}}} = \frac{V_{\text{sinal_analogico}}}{V_{1\text{bit}}}$$

Suponha que o formato do resultado seja justificado para a direita ($\text{ADFM} = 1$), então podemos representar o valor de tensão presente no canal analógico, no momento da conversão, pela equação a seguir.

$$V_{\text{sinal_analogico}} = V_{1\text{bit}} * \text{Valor}_{\text{conversor}}$$

13.2 Funções para o Módulo Conversor A/D

As funções de controle e configuração do módulo conversor A/D estão localizadas na biblioteca **adc.h**, e podem ser utilizadas em qualquer microcontrolador da família PIC18 que tenha esse módulo.

13.2.1 Verifica o Estado do Módulo

A função **BusyADC** verifica se o módulo conversor A/D está em um processo de conversão ou não, e retorna '1' se o periférico estiver ocupado, ou '0' se o periférico estiver livre para executar uma conversão.

Sintaxe

```
status = BusyADC();
```

Sendo:

- **status**: informa o status do conversor A/D. 1 - ocupado ou 0 - livre.

Exemplo

```
while( BusyADC() ); // Aguarda até que a conversão seja completada.  
resultado = ReadADC(); // Retorna o valor convertido.
```

13.2.2 Desabilita o Módulo

A função **CloseADC** desabilita o conversor A/D e a interrupção associada a ele.

Sintaxe

```
CloseADC();
```

Exemplo

```
CloseADC(); //Desabilita o módulo conversor A/D.
```

13.2.3 Inicia a Conversão A/D

A função **ConvertADC** inicia o processo de conversão do sinal analógico.

Sintaxe

```
ConvertADC();
```

Exemplo

```
ConvertADC(); //Inicia a conversão A/D.
```

```
while( BusyADC() ); // Aguarda até que a conversão seja completada.
```

```
resultado = ReadADC(); // Retorna o valor convertido.
```

13.2.4 Habilita o Módulo Conversor A/D

A função **OpenADC** habilita e configura o módulo conversor A/D.

Sintaxe

OpenADC (configuração_1 configuração_2, configuração_porta)

Sendo:

- **configuração_1**: são as constantes listadas na Tabela 13.3, 13.4 ou 13.5, separadas por um '&'.
- **configuração_2**: são as constantes listadas na Tabela 13.6, 13.7 ou 13.8 separadas por um '&'.
- **configuração_porta**: é uma constante listada na Tabela 13.9. (Válido apenas para os modelos listados nesta tabela)

Tabela 13.3: Constantes de configuração para os modelos PIC18CXX2, PIC18FXX2, PIC18FXX8, PIC18FXX39.

PIC18CXX2, PIC18FXX2, PIC18FXX8, PIC18FXX39		
Função	Constante	Descrição
Fonte de clock	ADC_FOSC_2	Fosc / 2
	ADC_FOSC_4	Fosc / 4
	ADC_FOSC_8	Fosc / 8
	ADC_FOSC_16	Fosc / 16
	ADC_FOSC_32	Fosc / 32
	ADC_FOSC_64	Fosc / 64
Formato do resultado	ADC_RIGHT_JUST	Resultado nos bits menos significativos.
	ADC_LEFT_JUST	Resultado nos bits mais significativos.
Tensão de referência	ADC_8ANA_0REF	$V_{ref+} = V_{DD}$ e $V_{ref-} = V_{SS}$. Todos os pinos são configurados como canal analógico. AN3 = V_{ref+}
	ADC_7ANA_1REF	$AN3 = V_{ref+}$ e $AN2 = V_{ref-}$
	ADC_6ANA_2REF	$V_{ref+} = V_{DD}$ e $V_{ref-} = V_{SS}$ AN3 = V_{ref+} e $V_{ref-} = V_{SS}$
	ADC_6ANA_0REF	$V_{ref+} = V_{DD}$ e $V_{ref-} = V_{SS}$ AN3 = V_{ref+} e $V_{ref-} = V_{SS}$
	ADC_5ANA_1REF	$V_{ref+} = V_{DD}$ e $V_{ref-} = V_{SS}$ AN3 = V_{ref+} e $AN2 = V_{ref-}$
	ADC_5ANA_0REF	$V_{ref+} = V_{DD}$ e $V_{ref-} = V_{SS}$ AN3 = V_{ref+} e $AN2 = V_{ref-}$
Formato do comando: ADC_xANA_yREF	ADC_4ANA_2REF	$AN3 = V_{ref+}$ e $AN2 = V_{ref-}$
	ADC_4ANA_1REF	$AN3 = V_{ref+}$
Sendo: x - N° de canais analógicos y - N° de V_{ref} externa	ADC_3ANA_2REF	$AN3 = V_{ref+}$ e $AN2 = V_{ref-}$
	ADC_3ANA_0REF	$V_{ref+} = V_{DD}$ e $V_{ref-} = V_{SS}$
	ADC_2ANA_2REF	$AN3 = V_{ref+}$ e $AN2 = V_{ref-}$
	ADC_2ANA_1REF	$AN3 = V_{ref+}$
	ADC_1ANA_2REF	$AN3 = V_{ref+}$ e $AN2 = V_{ref-}$
	ADC_1ANA_0REF	$AN0$ é entrada analógica.
	ADC_0ANA_0REF	Todos os pinos são configurados como I/O digital.

Tabela 13.4: Constantes de configuração para os modelos
PIC18C658/858, PIC18C601/801, PIC18F6X20, PIC18F8X20.

PIC18C658/858, PIC18C601/801, PIC18F6X20, PIC18F8X20		
Função	Constante	Descrição
Fonte de <i>clock</i>	ADC_FOSC_2	Fosc / 2
	ADC_FOSC_4	Fosc / 4
	ADC_FOSC_8	Fosc / 8
	ADC_FOSC_16	Fosc / 16
	ADC_FOSC_32	Fosc / 32
	ADC_FOSC_64	Fosc / 64
Formato do resultado	ADC_FOSC_RC	Oscilador RC interno.
	ADC_RIGHT_JUST	Resultado nos <i>bits</i> menos significativos.
Tensão de referência	ADC_LEFT_JUST	Resultado nos <i>bits</i> mais significativos.
	ADC_0ANA	Todos os pinos são configurados como I/O digital.
	ADC_1ANA	análogico:AN0 digital:AN1-AN15
	ADC_2ANA	análogico:AN0-AN1 digital:AN2-AN15
	ADC_3ANA	análogico:AN0-AN2 digital:AN3-AN15
	ADC_4ANA	análogico:AN0-AN3 digital:AN4-AN15
	ADC_5ANA	análogico:AN0-AN4 digital:AN5-AN15
	ADC_6ANA	análogico:AN0-AN5 digital:AN6-AN15
	ADC_7ANA	análogico:AN0-AN6 digital:AN7-AN15
	ADC_8ANA	análogico:AN0-AN7 digital:AN8-AN15
	ADC_9ANA	análogico:AN0-AN8 digital:AN9-AN15
	ADC_10ANA	análogico:AN0-AN9 digital:AN10-AN15
	ADC_11ANA	análogico:AN0-AN10 digital:AN11-AN15
	ADC_12ANA	análogico:AN0-AN11 digital:AN12-AN15
	ADC_13ANA	análogico:AN0-AN12 digital:AN13-AN15
	ADC_14ANA	análogico:AN0-AN13 digital:AN14-AN15
	ADC_15ANA	Todos os pinos são configurados como canal analógico.

Tabela 13.5: Constantes de configuração para os modelos
PIC18F1X20, PIC18F1X30, PIC18F2X20, PIC18F4X20,
PIC18FXX10/21/22/23/25/27/50/55/80/82/85/90 e PIC18FXXJ10/11/15/60/65/90

PIC18F1X20, PIC18F1X30, PIC18F2X20, PIC18F4X20, PIC18FXX10/21/22/23/25/27/50/55/80/82/85/90 e PIC18FXXJ10/11/15/60/65/90		
Função	Constante	Descrição
Fonte de <i>clock</i>	ADC_FOSC_2	Fosc / 2
	ADC_FOSC_4	Fosc / 4
	ADC_FOSC_8	Fosc / 8
	ADC_FOSC_16	Fosc / 16
	ADC_FOSC_32	Fosc / 32
	ADC_FOSC_64	Fosc / 64
	ADC_FOSC_RC	Oscilador RC interno.



**PIC18F1X20, PIC18F1X30, PIC18F2X20, PIC18F4X20,
PIC18FXX10/21/22/23/25/27/50/55/80/82/85/90 e PIC18FXXJ10/11/15/60/65/90**

Função	Constante	Descrição
Formato do resultado	ADC_RIGHT JUST ADC_LEFT JUST	Resultado nos <i>bits</i> menos significativos Resultado nos <i>bits</i> mais significativos
Tempo de aquisição automático	ADC_0_TAD	0 T_{AD}
	ADC_2_TAD	2 T_{AD}
	ADC_4_TAD	4 T_{AD}
	ADC_6_TAD	6 T_{AD}
	ADC_8_TAD	8 T_{AD}
	ADC_12_TAD	12 T_{AD}
	ADC_16_TAD	16 T_{AD}
	ADC_20_TAD	20 T_{AD}

Tabela 13.6: Constantes de configuração para os modelos
PIC18CXX2, PIC18FXX2, PIC18FXX8, PIC18FXX39.

PIC18CXX2, PIC18FXX2, PIC18FXX8, PIC18FXX39		
Função	Constante	Descrição
Seleção do canal	ADC_CH0	Canal 0 (AN0)
	ADC_CH1	Canal 1 (AN1)
	ADC_CH2	Canal 2 (AN2)
	ADC_CH3	Canal 3 (AN3)
	ADC_CH4	Canal 4 (AN4)
	ADC_CH5	Canal 5 (AN5)
	ADC_CH6	Canal 6 (AN6)
	ADC_CH7	Canal 7 (AN7)
Interrupção	ADC_INT_ON ADC_INT_OFF	Habilita. Desabilita.

Tabela 13.7: Constantes de configuração para os modelos
PIC18C658/858, PIC18C601/801, PIC18F6X20, PIC18F8X20.

PIC18C658/858, PIC18C601/801, PIC18F6X20, PIC18F8X20		
Função	Constante	Descrição
Seleção do canal	ADC_CH0	Canal 0 (AN0)
	ADC_CH1	Canal 1 (AN1)
	ADC_CH2	Canal 2 (AN2)
	ADC_CH3	Canal 3 (AN3)
	ADC_CH4	Canal 4 (AN4)
	ADC_CH5	Canal 5 (AN5)
	ADC_CH6	Canal 6 (AN6)
	ADC_CH7	Canal 7 (AN7)
	ADC_CH8	Canal 8 (AN8)
	ADC_CH9	Canal 9 (AN9)
Seleção do canal	ADC_CH10	Canal 10 (AN10)
	ADC_CH11	Canal 11 (AN11)
	ADC_CH12	Canal 12 (AN12)
	ADC_CH13	Canal 13 (AN13)
	ADC_CH14	Canal 14 (AN14)
	ADC_CH15	Canal 15 (AN15)
Interrupção	ADC_INT_ON	Habilita
	ADC_INT_OFF	Desabilita
Configuração da V_{ref+}	ADC_VREFPLUS_VDD	$V_{ref+} = AV_{CC}$
	ADC_VREFPLUS_EXT	$V_{ref+} = $ externo
Configuração da V_{ref-}	ADC_VREFMINUS_VSS	$V_{ref-} = AV_{SS}$
	ADC_VREFMINUS_EXT	$V_{ref-} = $ externo

Tabela 13.8: Constantes de configuração para os modelos
PIC18F1X20, PIC18F1X30, PIC18F2X20, PIC18F4X20,
PIC18FXX10/21/22/23/25/27/50/55/80/82/85/90 e PIC18FXXJ10/11/15/60/65/90

PIC18F1X20, PIC18F1X30, PIC18F2X20, PIC18F4X20, PIC18FXX10/21/22/23/25/27/50/55/80/82/85/90 e PIC18FXXJ10/11/15/60/65/90		
Função	Constante	Descrição
Seleção do canal	ADC_CH0	Canal 0 (AN0)
	ADC_CH1	Canal 1 (AN1)
	ADC_CH2	Canal 2 (AN2)
	ADC_CH3	Canal 3 (AN3)
	ADC_CH4	Canal 4 (AN4)
	ADC_CH5	Canal 5 (AN5)
	ADC_CH6	Canal 6 (AN6)
	ADC_CH7	Canal 7 (AN7)
	ADC_CH8	Canal 8 (AN8)
	ADC_CH9	Canal 9 (AN9)

**PIC18F1X20, PIC18F1X30, PIC18F2X20, PIC18F4X20,
PIC18FXX10/21/22/23/25/27/50/55/80/82/85/90 e PIC18FXXJ10/11/15/60/65/90**

Função	Constante	Descrição
Seleção do canal	ADC_CH10	Canal 10 (AN10)
	ADC_CH11	Canal 11 (AN11)
	ADC_CH12	Canal 12 (AN12)
	ADC_CH13	Canal 13 (AN13)
	ADC_CH14	Canal 14 (AN14)
	ADC_CH15	Canal 15 (AN15)
Interrupção	ADC_INT_ON	Habilita.
	ADC_INT_OFF	Desabilita.
Configuração da V_{ref}	ADC_VREFPLUS_VDD	$V_{ref+} = V_{CC}$
	ADC_VREFPLUS_EXT	$V_{ref+} = $ externo
	ADC_VREFMINUS_VSS	$V_{ref-} = V_{SS}$
	ADC_VREFMINUS_EXT	$V_{ref-} = $ externo

**Tabela 13.9 Constantes de configuração das portas do conversor A/D para os modelos
PIC18F1X20, PIC18F1X30, PIC18F2X20, PIC18F4X20,
PIC18FXX10/21/22/23/25/27/50/55/80/82/85/90 e PIC18FXXJ10/11/15/60/65/90**

**PIC18F1X20, PIC18F1X30, PIC18F2X20, PIC18F4X20,
PIC18FXX10/21/22/23/25/27/50/55/80/82/85/90 e PIC18FXXJ10/11/15/60/65/90**

Função	Constante	Descrição
Configuração das portas do conversor A/D	ADC_0ANA	Todos os pinos são configurados como I/O digital.
	ADC_1ANA	analógico:AN0 digital:AN1-AN15
	ADC_2ANA	analógico:AN0-AN1 digital:AN2-AN15
	ADC_3ANA	analógico:AN0-AN2 digital:AN3-AN15
	ADC_4ANA	analógico:AN0-AN3 digital:AN4-AN15
	ADC_5ANA	analógico:AN0-AN4 digital:AN5-AN15
	ADC_6ANA	analógico:AN0-AN5 digital:AN6-AN15
	ADC_7ANA	analógico:AN0-AN6 digital:AN7-AN15
	ADC_8ANA	analógico:AN0-AN7 digital:AN8-AN15
	ADC_9ANA	analógico:AN0-AN8 digital:AN9-AN15
	ADC_10ANA	analógico:AN0-AN9 digital:AN10-AN15
	ADC_11ANA	analógico:AN0-AN10 digital:AN11-AN15
	ADC_12ANA	analógico:AN0-AN11 digital:AN12-AN15
	ADC_13ANA	analógico:AN0-AN12 digital:AN13-AN15
	ADC_14ANA	analógico:AN0-AN13 digital:AN14-AN15
	ADC_15ANA	Todos os pinos são configurados como canal analógico.

Exemplo

```

#include <p18f2550.h> //Arquivo de cabeçalho do PIC18F2550.
#include <delays.h> //Adiciona a biblioteca de funções de atraso.
#include <adc.h> //Adiciona a biblioteca de funções para o módulo conversor A/D.

// Fosc = 10MHz
// Tciclo = 4/Fosc = 0,4us
#pragma config FOSC = HS
#pragma config CPUDIV = OSC1_PLL2

#pragma config WDT = OFF           //Desabilita o Watchdog Timer (WDT)
#pragma config PWRT = ON           //Habilita o Power-up Timer (PWRT).
#pragma config BOR = ON            //Brown-out Reset (BOR) habilitado somente no hardware.
#pragma config BORV = 1             //Voltagem do BOR é 4,33V.
#pragma config LVP = OFF           //Desabilita o Low Voltage Program.

#define LED LATCbits.LATC0 //Define outro nome para a estrutura

unsigned char buffer[16];

void main (void)
{
unsigned int resultado_conv;

OpenADC (ADC_FOSC_16
        &ADC_RIGHT JUST
        &ADC_2_TAD,
        ADC_CH0
        &ADC_INT_OFF
        &ADC_VREFPLUS_VDD
        &ADC_VREFMINUS_VSS,
        ADC_1ANA); //Fosc = 10MHz. Tad = 16/10M = 1,6us.
//Resultado justificado para a direita.
//Configuração do tempo de aquisição automático. (2*Tad = 3,2us)
//Seleciona o canal 0 (AN0)
//Interrupção desabilitada.
//Vref+ = Vcc
//Vref- = Vss
//Habilita somente o canal AN0

SetChanADC (ADC_CH0); //Seleciona o canal 0 (AN0).
Delay10TCYx( 5 ); // Delay de 50 ciclos de máquina.

TRISAbits.TRISA0 = 1; //Configura o pino AN0 como entrada.
TRISCbits.TRISC0 = 0; //Configura o pino RC0 como saída.
PORTCbits.RC0 = 0; //Desliga o LED.

while (1) //Looping infinito.
{
    ConvertADC ( ); //Inicia a conversão.
    while (BusyADC ( )); //Aguarda o fim da conversão.
    resultado_conv = ReadADC( ); //Armazena o resultado da conversão.

    // Vcc = 5 Volts.
    // Vref+ = Vcc e Vref- = Vss.
    // V_1bit = 4,887585.
    // resultado_conv = 410 -> V_sinal_analógico = 2 Volts.
    // Se o resultado_conv for maior que 410, o LED é ligado.
    if (resultado_conv >= 410)
        LED = 1; //Liga o LED.
    else
        LED = 0; //Desliga o LED.
}

```



13.2.5 Operação de Leitura

A função **ReadADC** retorna o resultado da conversão A/D.

Sintaxe

```
valor_16bits = ReadADC ( )
```

Sendo:

- **valor_16bits**: resultado da conversão.

Exemplo

```
unsigned int res_conv_AD;
```

```
while (BusyADC ( )); //Aguarda o fim da conversão.
```

```
res_conv_AD = ReadADC ( ); //Retorna o valor convertido.
```

13.2.6 Seleção do Canal Analógico

A função **SetChanADC** seleciona o canal usado como entrada para o conversor A/D.

Sintaxe

```
SetChanADC ( configuração )
```

Sendo:

- **configuração**: são as constantes listadas na Tabela 13.10, separadas por um '&'.

Tabela 13.10. Constantes de seleção do canal.

Constante	Descrição	Constante	Descrição
ADC_CH0	Canal 0 (AN0)	ADC_CH8	Canal 8 (AN8)
ADC_CH1	Canal 1 (AN1)	ADC_CH9	Canal 9 (AN9)
ADC_CH2	Canal 2 (AN2)	ADC_CH10	Canal 10 (AN10)
ADC_CH3	Canal 3 (AN3)	ADC_CH11	Canal 11 (AN11)
ADC_CH4	Canal 4 (AN4)	ADC_CH12	Canal 12 (AN12)
ADC_CH5	Canal 5 (AN5)	ADC_CH13	Canal 13 (AN13)
ADC_CH6	Canal 6 (AN6)	ADC_CH14	Canal 14 (AN14)
ADC_CH7	Canal 7 (AN7)	ADC_CH15	Canal 15 (AN15)

13.3 Projetos

Neste capítulo temos dois projetos propostos. O primeiro inclui a leitura da tensão regulada por um potenciômetro e sensor de temperatura (LM35DZ), e o segundo é o controle de um teclado analógico.

13.3.1 Leitura da Tensão Regulada por um Potenciômetro e Sensor de Temperatura

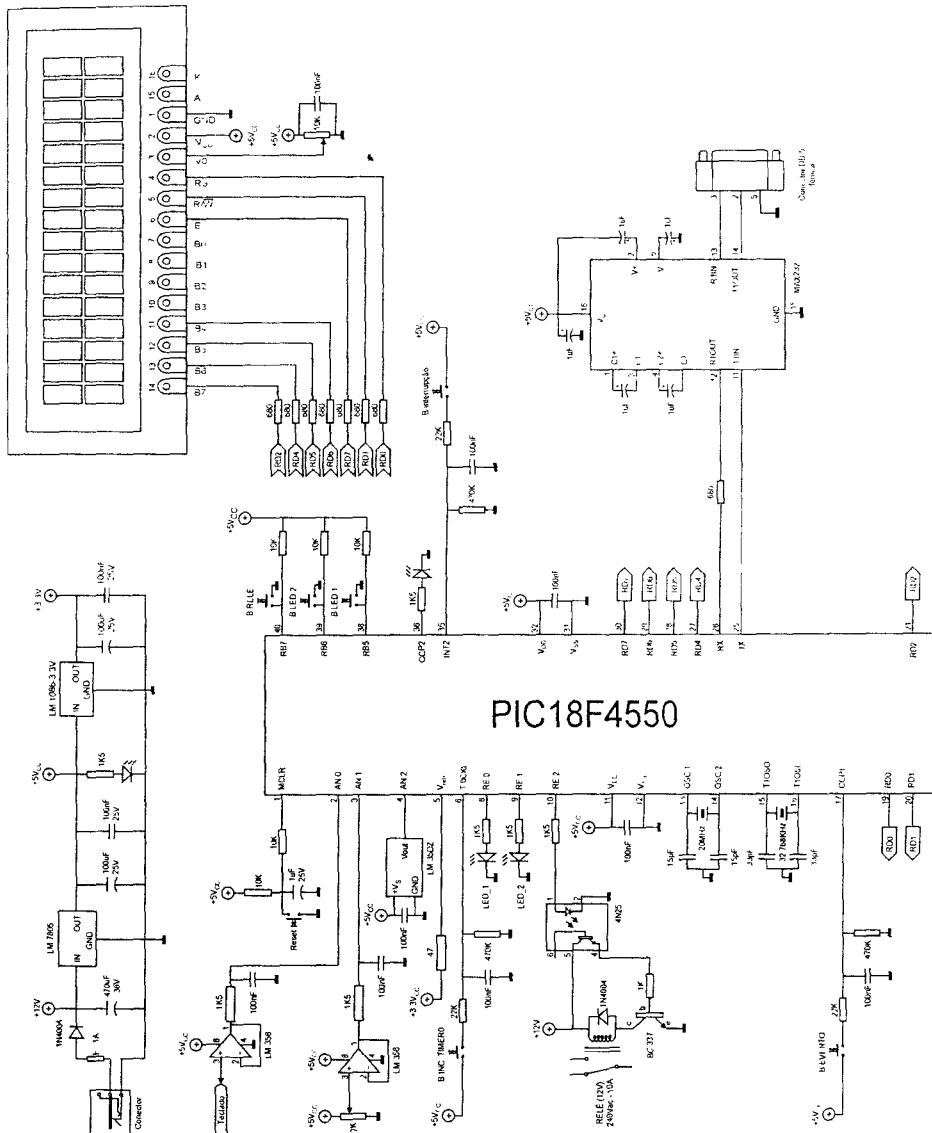


Figura 13.5: Circuito eletrônico para captura de sinal analógico.

Código

```
float valor_tensao, valor_temperatura; //Armazena o valor da tensão do potenciômetro e do LM35D.
unsigned char c_ponto = '.';
```

```
TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
TRISO = 0b00000000; //RD0 a RD7 – saída.
TRISE = 0b00000000; //RE0 a RE2 – saída.
```

```
OpenADC (ADC_FOSC_16
        &ADC_RIGHT_JUST    //Seleção da fonte de clock para a conversão A/D. Fosc = 20MHz. Tad = 0,8us.
        &ADC_4_TAD,         //Resultado justificado para a direita.
        ADC_CH1,           //Configuração do tempo de aquisição automático. (4*Tad = 3,2us)
        &ADC_INT_OFF,      //Seleciona o canal 1 (AN1)
        &ADC_VREFPLUS_EXT  //Interrupção desabilitada.
        &ADC_VREFMINUS_VSS, //Vref+ = Vref+ (pin AN3)
        &ADC_VREFMINUS_VSS, //Vref = Vss
        ADC_6ANA);          //Habilita o canal AN0 a AN5.
```

```
Delay10TCYx( 5 ); // Delay de 50 ciclos de máquina.
```

```
PORTE = 0x00; //Coloca a porta D em 0V.
PORTE = 0x00; //Coloca a porta E em 0V.
```

```
lcd_inicia(0x28, 0x0F, 0x06); //Inicializa o display LCD alfanumérico com quatro linhas de dados.
lcd_LD_cursor(0); //Desliga o cursor.
```

```
while (1) //Looping infinito.
{
    strcpypgm2ram(buffer,limpa_buffer); //Preenche o buffer com o caractere espaço ''.
    SetChanADC (ADC_CH1); //Carrega o canal analógico 1. (AN1) - Potenciômetro.
    valor = filtro_canal ( ); //Chama a função para fazer a leitura do canal.

    //Converte o valor retornado pelo conversor A/D em um valor de tensão correspondente.
    valor_tensao = converte_tensao(valor);

    lcd_posicao (1,1); // Desloca o ponteiro para a L=1 e C=1.
    sprintf(buffer,"Pot: %04ld mv", (long)valor_tensao); //Envia a string formatada para o buffer.
    imprime_buffer_lcd(buffer,12); //Coloca no visor, os 12 elementos do buffer.

    strcpypgm2ram(buffer,limpa_buffer); //Preenche o buffer com o caractere espaço ''.
    SetChanADC (ADC_CH2); //Carrega o canal analógico 2. (AN2) - Sensor de temperatura LM35D.
    valor = filtro_canal ( ); //Chama a função para fazer a leitura do canal.

    //Converte o valor retornado pelo conversor A/D em um valor de tensão correspondente.
    valor_tensao = converte_tensao(valor);
    valor_temperatura = converte_temperatura (valor_tensao); //Converte a tensão em temperatura.

    lcd_posicao (2,1); // Desloca o ponteiro para a L=2 e C=1.
    sprintf(buffer,"Temp: %03d.%02d C", (char)valor_temperatura,(char)((valor_temperatura-(char)valor_temperatura)*100));
    //Envia a string formatada para o buffer.
    imprime_buffer_lcd(buffer,14); //Coloca no visor, os 14 elementos do buffer.

    Delay10KTCYx (100); //Gera um delay de 200 milissegundos. 100*10.000*0,2us= 200ms
}
```

13.3.2 Teclado Analógico

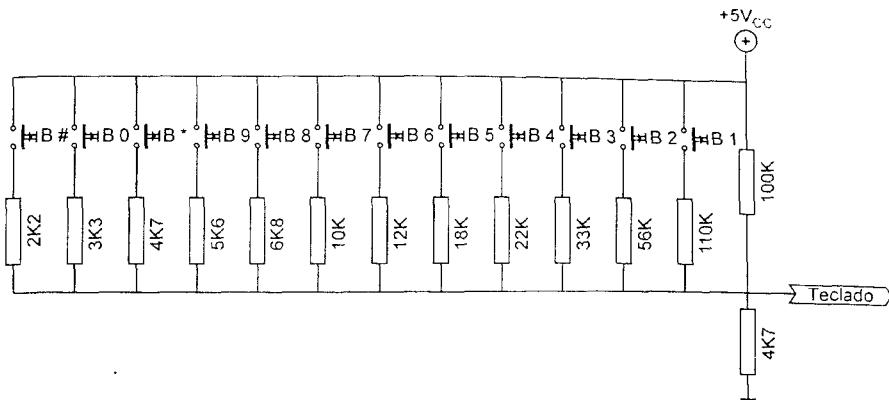


Figura 13.6: Circuito eletrônico do teclado analógico.

A Tabela 13.11 lista os valores de tensão relacionados aos botões do teclado.

Tabela 13.11: Relação entre botão pressionado x tensão de saída.

Tensão de saída, conforme o botão pressionado					
B 1	B 2	B 3	B 4	B 5	B 6
0.416V	0.579V	0.796V	1.034V	1.178V	1.525V
B 7	B 8	B 9	B *	B 0	B #
1.704V	2.123V	2.349V	2.557V	2.977V	3.429V

Observação: Em repouso o valor de saída é 0.224V

Código

```
***** Projeto Capítulo 13 - TECLADO ANALÓGICO *****
**
** Este programa mostra no display LCD 2x16, o caractere correspondente ao último botão pressionado.
**
** Autor: Alberto Noboru Miyadaira
*****
```

```
#include <p18f4550.h>          //Arquivo de cabeçalho do PIC18F4550.
#include <delays.h>            //Adiciona a biblioteca de funções de atraso.
#include <adc.h>                //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <stdio.h>              //Adiciona a biblioteca padrão de entrada e saída.
#include <stdlib.h>              //Adiciona a biblioteca de funções miscelâneas.
#include <string.h>              //Adiciona a biblioteca de manipulação de string.
#include "C:\pic18\oteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)
```

```
//Variáveis globais.
unsigned int validade_tecla, cont_tecla;
unsigned char tecla, tecla_anterior;
unsigned char buffer[16];
unsigned char cont_t;
```

```

unsigned int valor; // Armazena o valor devolvido pelo conversor A/D.
float valor_tensao; // Armazena o valor da tensão do teclado.
//Matriz contendo os ranges de tensão
unsigned int range_tensao [13]={300, 500, 700, 900, 1100, 1400, 1650, 1900, 2200, 2450, 2700, 3200, 3600};

//Retorna o caractere correspondente ao número pressionado.
unsigned char nome_tecla(unsigned char tecla)
{
//Observação: não há necessidade do comando break, pois o comando return força a saída da função.

    switch (tecla)
    {
        case 1: return '1';
        case 2: return '2';
        case 3: return '3';
        case 4: return '4';
        case 5: return '5';
        case 6: return '6';
        case 7: return '7';
        case 8: return '8';
        case 9: return '9';
        case 10: return '*';
        case 11: return '0';
        case 12: return '#';
        default: return 0;
    }
}

//Filtro em software.
//Obtém 32 amostras e retorna a média.
float filtro_canal()
{
    unsigned int cont_filtro;
    unsigned long valor_canal = 0;

    for( cont_filtro = 0 ; cont_filtro < 32 ; cont_filtro++ )
    {
        ConvertADC (); //Inicia a conversão.
        while (BusyADC ()); //Aguarda o fim da conversão.
        valor_canal += ReadADC(); //Armazena o resultado da conversão
    }
    return (valor_canal >> 5); // Esta operação é igual a valor_canal/32.
}

//Converte o valor devolvido pelo conversor, em um valor correspondente a tensão [mV].
float converte_tensao (float valor_conversor)
{
    return (valor_conversor*5000)/1023;//Neste caso, a tensão de referência é 5V.
}

//Retorna o status do botão informado.
unsigned char teclado (unsigned char valor)
{
    valor_tensao = converte_tensao(filtro_canal ());

    if (valor_tensao >= range_tensao[cont_t-1] && valor_tensao < range_tensao[cont_t])
        return 1;
    else
        return 0;
}

//Filtro implementado em software, verifica 30 amostras.
unsigned char filtro_teclado (unsigned char num_tecla)

```

```

{
    validade_tecla=0;

    for( cont_tecla = 0; cont_tecla < 30 ; cont_tecla++ )
    {
        validade_tecla = validade_tecla + teclado( num_tecla );
    }

    if(validade_tecla > 24)
        return num_tecla;
    else
        return 0;
}

//Atribui um valor ao botão, e faz uma verificação da validade dele.
unsigned char checa_tecla(unsigned char desc_tecla_ant)
{
    tecla=0;

    if (desc_tecla_ant == 0)//Desconsidera a tecla anterior.
    tecla_anterior = 0;

    valor_tensao = converte_tensao(filtro_canal ()). //Chama a função para fazer a leitura do canal.

    if(valor_tensao > 300)
    {
        for(cont_t=1; cont_t<13; cont_t++)
        {
            if (valor_tensao >= range_tensao[cont_t-1] && valor_tensao< range_tensao[cont_t])
            {
                if( tecla_anterior != cont_t )
                    tecla = filtro_teclado(cont_t);
                else
                    tecla = cont_t.

                break; //Sai do laço for.
            }
        }
    }

    if (tecla_anterior == tecla) //Se for a mesma tecla, então retorna 0
        tecla = 0;
    else //Caso contrário retorna o número correspondente à tecla
        tecla_anterior = tecla;

    return nome_tecla (tecla); //Retorna o caractere correspondente à tecla pressionada.
}

```

```

void main (void)
{
//Variáveis locais.
unsigned int valor; // Armazena o valor devolvido pelo conversor A/D.
float valor_tensao;
unsigned char botao_pres; //Número do botão.

```

```

TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
TRISD = 0b00000000; //RD0 a RD7 – saída
TRISE = 0b00000000; //RE0 a RE2 – saída.

```

OpenADC (ADC_FOSC_16 //Seleção da fonte de clock para a conversão A/D. Fosc = 20MHz. Tad = 0,8us.
&ADC_RIGHT JUST //Resultado justificado para a direita.

```

    &ADC_4_TAD,           //Configuração do tempo de aquisição automática. (4*Tad = 3,2us)
    ADC_CH0,              //Seleciona o canal 0 (AN0)
    &ADC_INT_OFF,         //Interrupção desabilitada
    &ADC_VREFPLUS_VDD,   //Vref+ = Vcc (Tensão de alimentação)
    &ADC_VREFMINUS_VSS,  //Vref- = Vss
    ADC_ANA);             //Habilita o canal AN0 a AN5.

```

Delay10TCYx(5); // Delay de 50 ciclos de máquina.

PORTE = 0x00; //Coloca a porta D em 0V.
 PORTE = 0x00; //Coloca a porta E em 0V.

lcd_inicia(0x28, 0x0F, 0x06); // Inicializa o display LCD alfanumérico com quatro linhas de dados.
 lcd_LD_cursor(0); // Desliga o cursor.

lcd_posicao(1,1); // Desloca o ponteiro para a L=1 e C=1.

imprime_string_lcd("Teclado Analog."); //Imprime uma string no display.

while (1) //Looping infinito.

{
 botao_pres = checa_tecla(1); //Verifica se existe algum botão pressionado.

if(botao_pres != 0)

{
 lcd_posicao(2,1); // Desloca o ponteiro para a L=2 e C=1.
 //Mostra o símbolo correspondente ao botão pressionado.

sprintf(buffer, "Teclado: %c",botao_pres); //Envia a string formatada para o buffer.

imprime_buffer_lcd(buffer,10); //Coloca no visor os dez primeiros elementos do buffer.

}

Delay10KTCYx (100); //Gera um delay de 200 milissegundos. 100*10.000*0,2us= 200ms

}

14

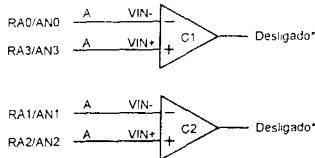
Módulo Comparador Analógico e de Tensão de Referência

Este capítulo comprehende dois módulos: módulo comparador analógico e módulo de tensão de referência (CV_{ref}). Ambos são tratados em um mesmo capítulo, pois o módulo de tensão de referência complementa o outro, no entanto pode ser tratado de forma independente.

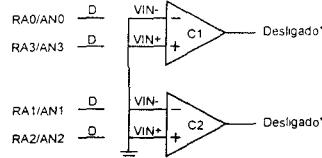
14.1 Módulo Comparador

O módulo comparador analógico presente no PIC18F4550 é constituído de dois comparadores, cujo modo de operação pode ser configurado de acordo com a combinação dos bits CM2:CM0 (**CMCON<2:0>**). Veja a Figura 14.1.

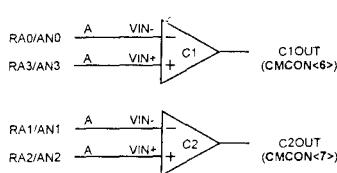
Reset do comparador
CM2 CM0 = 000



Comparador desligado
CM2 CM0 = 111



Dois comparadores independentes
CM2 CM0 = 010



Dois comparadores independentes com sinal de saída
CM2 CM0 = 011

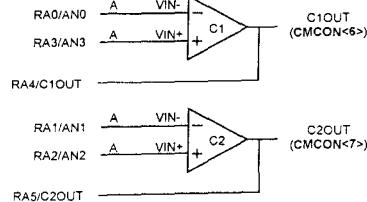
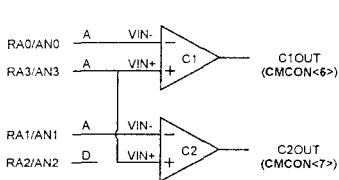
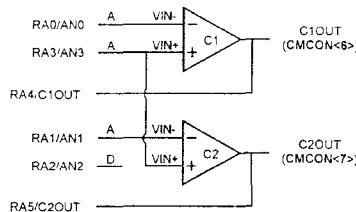


Figura 14.1: Modos de operação do módulo comparador do PIC18F4550. (continua)

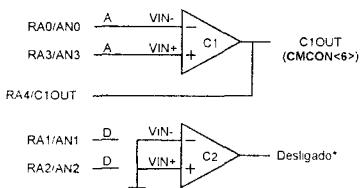
Dois comparadores com a mesma referência
CM2 CM0 = 100



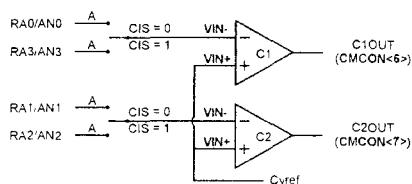
Dois comparadores com a mesma referência e com sinal de saída
CM2 CM0 = 101



Um comparador independente com sinal de saída
CM2 CM0 = 001



Quatro entradas multiplexadas para dois comparadores
CM2 CM0 = 110



A = Entrada analógica

D = Entrada digital

CIS (CMCON<3>) = Chave de entrada do comparador

* - Quando o comparador está desligado, o valor do registro (C1OUT e/ou C2OUT) é igual a 0.

Figura 14.1 Modos de operação do módulo comparador do PIC18F4550.



Se os pinos RA4/C1OUT e RA5/C2OUT estiverem habilitados, eles devem ser configurados como saída através do registro TRIS.

Note que na configuração CM2:CM0 = 110 os pinos de entrada do comparador são multiplexados, e a seleção é feita através do bit CIS (CMCON<3>). Sendo:

- CIS = 0: C1 – RA0/AN0 → VIN– e C2 - RA1/AN1 → VIN–.
- CIS = 1: C1 – RA3/AN3 → VIN– e C2 - RA2/AN2 → VIN–.

Neste modo de configuração, o pino VIN+ de ambos os módulos está ligado a uma tensão de referência gerada internamente (CV_{ref}). Veremos como configurá-la no próximo tópico.

A saída do módulo comparador analógico (C1OUT ou C2OUT) pode operar em dois modos distintos: saída não invertida ou saída invertida, cuja seleção é realizada pelos bits C1INV (CMCON<4>) e C2INV (CMCON<5>), sendo '0' para o modo não invertido e '1' para o modo invertido.

Comparador 1	
C1INV = 0	C1INV = 1
 Se $VIN_- > VIN_+$, logo, C1OUT = 0 Se $VIN_- < VIN_+$, logo, C1OUT = 1	 Se $VIN_- > VIN_+$, logo, C1OUT = 1. Se $VIN_- < VIN_+$, logo, C1OUT = 0.

Figura 14.2: Modo de operação da saída do comparador 1.

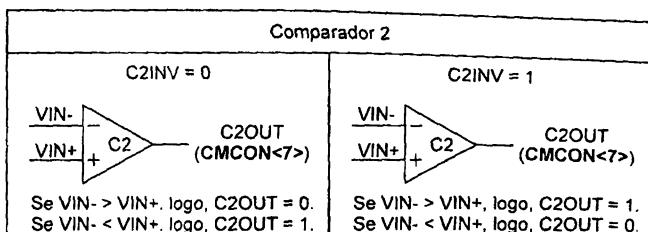


Figura 14.3: Modo de operação da saída do comparador 2.



A Microchip recomenda que a resistência de entrada no pino do comparador seja menor que $10\text{K}\Omega$.

Quando o módulo está habilitado, qualquer alteração na saída do comparador 1 ou 2 vai setar o *Flag bit* de interrupção CMIF (PIR2<6>). Se esse evento ocorrer enquanto o *bit* CMIE (PIE2<6>) estiver setado, uma interrupção pode ser gerada.

14.2 Módulo de Tensão de Referência

A tensão de referência do módulo comparador (CV_{ref}) é fornecida pelo módulo de tensão de referência, sendo este habilitado/desabilitado através do *bit* CVREN (CVRCON<7>).

- **CVREN = 0:** desabilita o módulo de tensão de referência.
- **CVREN = 1:** habilita o módulo de tensão de referência.

Veja a seguir o diagrama de blocos do módulo de tensão de referência.

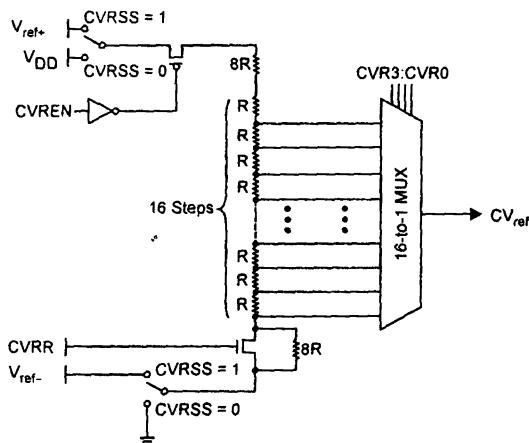


Figura 14.4: Diagrama de blocos do módulo de tensão de referência.

Na Figura 14.4 é possível verificar que a saída de tensão de referência (CV_{ref}) depende da tensão de alimentação (CVRSS (CVRCON<4>)) do módulo e da escolha de uma das duas faixas de tensão de saída, selecionadas pelo *bit* CVRR (CVRCON<5>), sendo cada uma de 16 níveis (CVR3:CVR0 (CVRCON<3:0>)).

A fonte de tensão é selecionada pelo bit CVRSS, sendo:

- **CVRSS = 0:** $CV_{RSRC} = V_{CC} - V_{SS}$.
- **CVRSS = 1:** $CV_{RSRC} = V_{ref+} - V_{ref-}$.

Enquanto a seleção da faixa de tensão depende do bit CVRR.

- **CVRR = 1:** faixa de tensão de 0 a 0,667 CV_{RSRC} com tamanho do passo igual a $CV_{RSRC}/24$.
- **CVRR = 0:** faixa de tensão de 0,25 CV_{RSRC} a 0,75 CV_{RSRC} com tamanho do passo igual a $CV_{RSRC}/32$.

Após configurar a fonte de tensão (CVRSS) e a faixa de tensão (CVRR), pode-se definir a tensão de saída (CV_{ref}) de acordo com as duas condições a seguir.

Para $CVRR = 1$, temos a seguinte equação:

$$CV_{ref} = \frac{(CVR3 : CVR0) * CV_{RSRC}}{24}$$

E para $CVRR = 0$, temos:

$$CV_{ref} = \frac{CV_{RSRC}}{4} + \frac{(CVR3 : CVR0) * CV_{RSRC}}{32}$$

Existe também a possibilidade de colocar a tensão de referência fornecida por esse módulo (CV_{ref}) na saída do pino $CV_{ref}/RA2/AN2/V_{ref-}$ pela configuração do bit CVROE (CVRCON<6>).

- **CVROE = 1:** nível de tensão CV_{ref} conectado ao pino CV_{ref-} .
- **CVROE = 0:** nível de tensão CV_{ref} desconectado do pino CV_{ref-} .

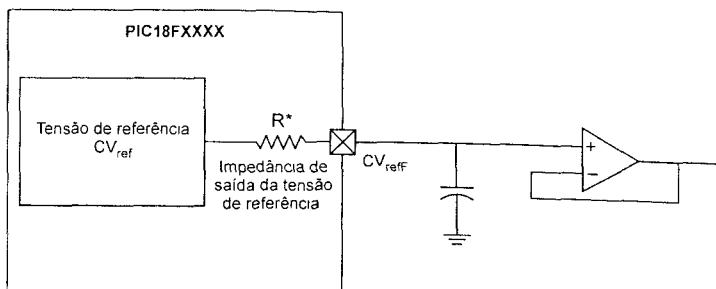
Se o nível de tensão CV_{ref} estiver conectado ao pino de saída CV_{ref} (CVROE = 1), a resistência de saída desse pino depende da combinação dos bits CVRR e CVR3:CVR0.

Como dito no início do capítulo, o módulo de tensão de referência pode operar de modo independente, podendo ser utilizado como um simples conversor D/A, com capacidade limitada, ou como saída de tensão de referência. Contudo, a capacidade de corrente desse módulo é bastante limitada, sendo necessário adicionar um buffer de tensão.

Um buffer de tensão possui as seguintes características:

- Alta impedância de entrada.
- Baixa impedância de saída.
- Ganho unitário.

Ele é utilizado com a finalidade de fazer um casamento de impedância e ganho de potência, uma vez que a corrente de saída do buffer depende exclusivamente de sua capacidade de corrente, e não mais do pino CV_{ref} . Veja a seguir o buffer de tensão sugerido pela Microchip.



Legenda * - A resistência é afetada pelos bits de configuração, CVRR e CVR3 CVR0

Figura 14.5: Exemplo de *buffer* de tensão.

14.3 Projeto

Este projeto utiliza o circuito eletrônico apresentado no Capítulo 13.

Código

```
***** Projeto Capítulo 14 - COMPARADOR *****
**
** Este projeto aciona o LED_1 caso o valor do potenciômetro (AN1) for superior a tensão de referência
**
** Autor: Alberto Noboru Miyadaira
***** */

#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550
#include <adc.h> //Adiciona a biblioteca de funções para o módulo conversor A/D
#include "C:\pic18\ biblioteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)

void ISR_alta_prioridade(void); //Protótipo da função de interrupção.
#define LED_1 LATEnbits.LATE0 //Define outro nome para a estrutura.

#pragma code int_alta=0x08 //Vetor de interrupção de alta prioridade, ou padrão.
void int_alta (void)
{
    _asm GOTO ISR_alta_prioridade _endasm //Desvia o programa para a função ISR_alta_prioridade.
}
#pragma code

#pragma interrupt ISR_alta_prioridade
void ISR_alta_prioridade(void)
{
    //Se o valor do potenciômetro for maior que CVref, a rotina dentro do if é executada.
    if (CMCONbits.C2OUT == 1)
        LED_1 = 1; //Liga o LED_1.
    else
        LED_1 = 0; //Desliga o LED_1.

    PIR2bits.CMIF = 0; // Limpa o flag bit da interrupção do módulo comparador.
}

void main (void)
{
    TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
```

```

TRISD = 0b00000000; //RD0 a RD7 – saída.
TRISE = 0b00000000; //RE0 a RE2 – saída.

//Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital. (Capítulo 13)
//Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h.
CloseADC (); //Requer a biblioteca adc.h.

PORTD = 0x00; //Coloca a porta D em 0V.
PORTE = 0x00; //Coloca a porta E em 0V.

//Quatro entradas multiplexadas para dois comparadores.
CMCONbits.CM2 = 1;
CMCONbits.CM1 = 1;
CMCONbits.CM0 = 0;

CMCONbits.CIS = 0; //RA0/AN0 conectado no pino VIN- do Comparador 1.
//RA1/AN1 conectado no pino VIN- do Comparador 2.
//CVref conectado no pino VIN+ dos dois Comparadores.

CMCONbits.C2INV = 1;//Saída do Comparador 2 é invertida.
//Se VIN- > VIN+ --> C2OUT = 1.
//Se VIN- < VIN+ --> C2OUT = 0.

CVRCONbits.CVRSS = 0; //CVsrc = Vcc – Vss = 5V.

CVRCONbits.CVR3 = 1;
CVRCONbits.CVR2 = 0;
CVRCONbits.CVR1 = 1;
CVRCONbits.CVR0 = 0; //CVR3:CVR0 = 10

// CVref = CVsrc/4 + ((CVR3:CVR0)*CVsrc)/32 = 5/4 + (10^5)/32 = 2,8125V
CVRCONbits.CVRR = 0;

CVRCONbits.CVROE = 0; //O nível de tensão CVref é desconectado do pino CVref
CVRCONbits.CVREN = 1; //Habilita o módulo de tensão de referência.

PIR2bits.CMIF = 0; //Limpa o flag bit da interrupção do módulo comparador.
IPR2bits.CMIP = 1; //Prioridade alta.
PIE2bits.CMIE = 1; //Ativa a interrupção do módulo comparador.

RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo.
INTCONbits.GIEH = 1; //Habilita todas as interrupções de alta prioridade.
INTCONbits.GIEL = 1; //Habilita todas as interrupções de baixa prioridade.

lcd_inicia(0x28, 0x0F, 0x06); //Inicializa o display LCD alfanumérico com quatro linhas de dados.
lcd_LD_cursor (0); //Desliga o cursor.

lcd_posicao(1,1); //Posiciona o cursor na L=1 e C=1;
imprime_string_lcd( "CVref = 2.8125V"); //Imprime a string
while (1); //Looping infinito.
}

```

Comunicação I²C

O I²C (*Inter-Integrated Circuit*) é um protocolo de comunicação síncrono Mestre-Escravo (*Master-Slave*), criado pela PHILIPS, com a finalidade de simplificar a comunicação entre dispositivos eletrônicos. Atualmente, ele é largamente difundido e adotado por diversos fabricantes que o utilizam em seus produtos, dentre eles driver de LCD, memória serial, gerador de tom, conversor A/D e D/A, RTC (*Real Time Clock*), microcontrolador etc.

O barramento I²C é composto por duas linhas de comunicação, sendo uma linha de dado (SDA) e uma de clock (SCL). A linha SCL é controlada pelo dispositivo configurado como *Master*, o qual é responsável pelo envio do sinal de *clock* para o dispositivo *Slave*, enquanto a linha SDA é bidirecional, e carrega a informação (8bits).

Esse barramento é do tipo *Multi-Master*, o que significa que mais de um dispositivo pode iniciar uma comunicação (enviar o sinal de *clock*), porém todos os outros dispositivos devem estar configurados como *Slave* durante esse processo. Geralmente, os dispositivos configurados como *Master* são microcontroladores ou microprocessadores devido a sua capacidade de processamento.

Cada dispositivo conectado no barramento possui um endereço único, representado por 7 ou 10bits, podendo atuar como transmissor ou receptor.

A Figura 15.1 mostra um exemplo de como os dispositivos podem ser conectados no barramento I²C.

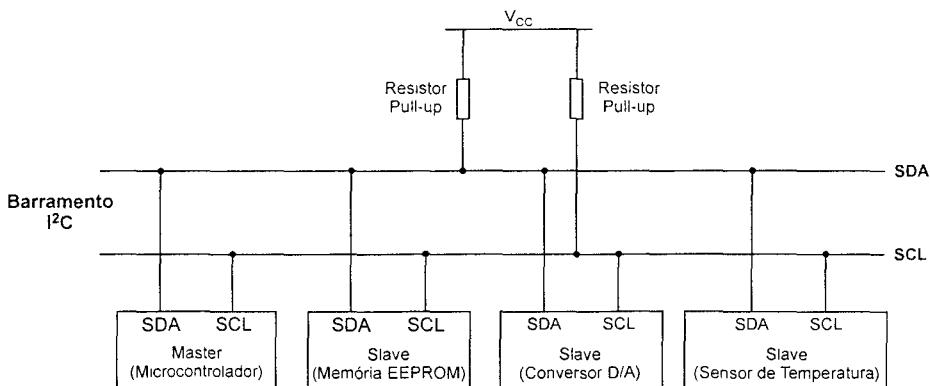


Figura 15.1: Barramento I²C.

Observe que existem resistores de *pull-up* conectados nas linhas SCL e SDA. Eles são essenciais para o funcionamento do barramento I²C, pois indicam que o barramento está disponível, uma vez que os dispositivos conectados a ele possuem apenas a função de abaixar a tensão. Esses resistores estão normalmente localizados na faixa de 1K a 10K, e devem ser inseridos no barramento, a fim de evitar que as linhas fiquem flutuando, e com isso prevenir eventuais erros de comunicação.

Outro modo de evitar erros de comunicação é garantir que a frequência de *clock* gerada pelo *Master* não ultrapasse os limites de frequência dos dispositivos escravos. Geralmente a taxa de transferência é padronizada: 100Kbps, 400Kbps e 1Mbps.

15.1 Funcionamento do Protocolo I²C

O inicio e o término de uma comunicação I²C são identificados por duas condições: inicio (START) e parada (STOP). A condição de START é verificada quando a linha de dado (SDA) passa do nível lógico '1' para o nível '0', enquanto a linha de *clock* (SCL) está em nível '1'. Uma condição de STOP é identificada quando a linha SDA passa do nível '0' para o nível '1', enquanto a linha de SCL se encontra em nível lógico '1'.

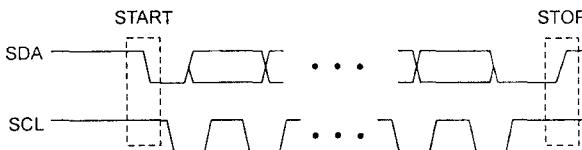


Figura 15.2. Condição de START e STOP.

Até o presente momento, foram apresentadas as condições de START e STOP de uma comunicação I²C. Veremos na sequência como se procede à comunicação entre um dispositivo *Master* e um *Slave*.

Na comunicação com endereçamento de 7bits, o dispositivo *Master* envia uma condição de START, seguida de 8bits, que compreende o endereço do dispositivo *Slave* (indicado pela letra 'A') e o tipo de operação (R/W), sendo:

- **R/W = 1:** operação de leitura.
- **R/W = 0:** operação de escrita.

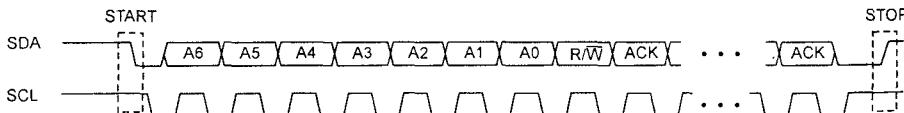


Figura 15.3: Comunicação com endereço de 7bits.

A comunicação entre o dispositivo *Master* e o *Slave* com endereço de 10bits é apresentada a seguir.

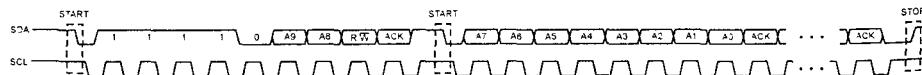


Figura 15.4: Comunicação com endereço de 10bits.

No modo de endereçamento de 10bits, os 5bits mais significativos do primeiro pacote (8bits) são constantes, pois indicam aos dispositivos *Slaves* que o modo de endereçamento será de 10bits.

Dando sequência ao raciocínio, vamos considerar como exemplo o modo de endereçamento de 7bits. Após o envio do endereço e o tipo de operação, os dispositivos *Slave* verificam se o endereço solicitado pelo *Master* pertence a um deles e retornam um sinal de *Acknowledge* (força a linha SDA para o nível '0' antes do nono pulso de *clock* - ACK = 0) caso seja reconhecido; caso contrário, simplesmente não respondem (*Not Acknowledge* - ACK = 1). A descrição do bit ACK pode ser vista logo a seguir.

- ACK = 0: dispositivo/comando/dado reconhecido.
- ACK = 1: dispositivo/comando/dado não reconhecido.

Se nenhum dispositivo *Slave* responder à chamada, o dispositivo *Master* deve enviar uma condição de STOP para indicar o fim da comunicação. Todavia, se a chamada for atendida por algum dispositivo *Slave*, ele responde de acordo com a operação solicitada (R/W).



Figura 15.5: Protocolo I²C, endereço de 7bits.

Se R/W = 1, o dispositivo *Master* envia oito pulsos de *clock*, e a cada pulso efetua a leitura do nível lógico ('0' ou '1') presente na linha SDA, realizando então, a leitura dos 8bits enviados pelo *Slave*, iniciando pelo bit mais significativo. Ao receber o oitavo bit, o dispositivo *Master* deve responder, enviando um nono pulso com ACK = 0 (linha SDA com nível lógico '0').

Se R/W = 0, o dispositivo *Master* envia oito pulsos de *clock*, carregando o dado (8bits) na linha SDA, e após o oitavo bit, o dispositivo *Slave* deve responder com ACK = 0 (linha SDA com nível lógico '0').

Durante a operação de escrita/leitura, um ou mais dados podem ser enviados/recebidos tanto para o dispositivo *Master* como para o *Slave*, e a comunicação é finalizada com um comando de STOP.

15.2 Comunicação I²C do PIC18F4550

A interface I²C do PIC18F4550 está implementada no módulo *Master Synchronous Serial Port* (MSSP) e pode operar no modo *Master*, *Multi-Master* ou *Slave* com suporte a chamada geral (endereço:operação - 0000000:0).

Os pinos usados para transferência de dados são:

- **SCL**: pino de envio ou recebimento do sinal de *clock*.
- **SDA**: pino de transferência de dados.



O sentido de fluxo dos pinos deve ser definido pelo registro **TRIS**. Se o dispositivo estiver configurado como *Master*, o pino SCL deve ser definido como saída (**TRIS<1>** = 0); caso contrário, deve ser configurado como entrada (**TRIS<1>** = 1).

15.2.1 Registro de Endereço/Baud Rate (SSPADD)

O registro **SSPADD** possui duas funções. Quando o dispositivo está configurado como *Slave*, ele contém o endereço do dispositivo (7bits menos significativos), e quando configurado como *Master*, ele armazena o valor que define a taxa de comunicação (*Baud rate*) do I²C, o qual está restrito aos 7bits menos significativos do registro **SSPADD**.

$$\text{SSPADD } < 6 : 0 > = \frac{\text{Fosc}}{4 * \text{Baud rate}} - 1$$



As frequências padronizadas pelo protocolo I²C são 100KHz e 400KHz.

15.3 Funções de Controle/Configuração do Periférico I²C

As funções de controle e configuração do periférico I²C dos microcontroladores da família PIC18 estão implementadas na biblioteca i2c.h. Todas as funções que serão apresentadas na sequência possuem três formatos. Por exemplo: **AckI2C** deve ser usada em dispositivos que possuam apenas um periférico I²C, enquanto **AckI2C1** ou **AckI2C2** devem ser usadas em dispositivos com mais de um periférico I²C.

15.3.1 Condição de Acknowledge (ACK)

A função **AckI2C** gera uma condição de *Acknowledge*.

Sintaxe

AckI2C ()
AckI2C1 ()
AckI2C2 ()

15.3.2 Condição de Not Acknowledge (Not ACK)

A função **NotAckI2C** gera uma condição de *Not Acknowledge*.

Sintaxe

NotAckI2C ()
NotAckI2C1 ()
NotAckI2C2 ()

15.3.3 Condição de RESTART

A função **RestartI2C** gera uma condição de RESTART no barramento I²C.

Sintaxe

RestartI2C ()
RestartI2C1 ()
RestartI2C2 ()

Exemplo

RestartI2C ();	//Gera uma condição de RESTART no barramento I ² C do módulo SSP.
RestartI2C1 ();	//Gera uma condição de RESTART no barramento I ² C do módulo SSP 1.
RestartI2C2 ();	//Gera uma condição de RESTART no barramento I ² C do módulo SSP 2.

15.3.4 Condição de START

A função **StartI2C** gera uma condição de START no barramento I²C.

Sintaxe

StartI2C ()
StartI2C1 ()
StartI2C2 ()

Exemplo

```
StartI2C();           //Gera uma condição de START no barramento I2C do módulo SSP
StartI2C1();          //Gera uma condição de START no barramento I2C do módulo SSP 1.
StartI2C2();          //Gera uma condição de START no barramento I2C do módulo SSP 2.
```

15.3.5 Condição de STOP

A função **StopI2C** gera uma condição de STOP no barramento I²C.

Sintaxe

```
StopI2C()
StopI2C1()
StopI2C2()
```

Exemplo

```
StopI2C();           //Gera uma condição de STOP no barramento I2C do módulo SSP.
StopI2C1();          //Gera uma condição de STOP no barramento I2C do módulo SSP 1.
StopI2C2();          //Gera uma condição de STOP no barramento I2C do módulo SSP 2.
```

15.3.6 Configura o Periférico I²C

A função **OpenI2C** configura o periférico I²C pertencente ao módulo SSP.

Sintaxe

```
OpenI2C( modo, velocidade )
OpenI2C1( modo, velocidade )
OpenI2C2( modo, velocidade )
```

Sendo:

- **modo**: uma das constantes listadas na Tabela 15.1.
- **velocidade**: uma das constantes listadas na Tabela 15.2.

Tabela 15.1: Define o modo de operação da interface I²C.

Constante	Descrição
SLAVE_7	Modo Slave com endereço de 7bits.
SLAVE_10	Modo Slave com endereço de 10bits.
MASTER	Modo Master.

Tabela 15.2: Tratamento de borda da interface I²C.

Constante	Descrição
SLEW_OFF	Tratamento de borda desabilitado. Frequência de 100KHz e 1MHz.
SLEW_ON	Tratamento de borda habilitado. Frequência de 400KHz.

Exemplo

```
OpenI2C(MASTER, SLEW_ON);           //Configura o periférico I2C do módulo SSP.
OpenI2C1(SLAVE_10, SLEW_ON);         //Configura o periférico I2C do módulo SSP 1.
OpenI2C2(MASTER, SLEW_OFF);          //Configura o periférico I2C do módulo SSP 2.
```

15.3.7 Desabilita o Periférico I²C

A função **Closel2C** desabilita o periférico I²C pertencente ao módulo SSP.

Sintaxe

```
Closel2C()
Closel2C1()
Closel2C2()
```

Exemplo

```
Closel2C(); // Desabilita o periférico I2C do módulo SSP.
Closel2C1(); // Desabilita o periférico I2C do módulo SSP 1.
Closel2C2(); // Desabilita o periférico I2C do módulo SSP 2.
```

15.3.8 Recepção de Caractere

A função **ReadI2C** ou **getcl2C** lê um único byte vindo do barramento I²C.

Sintaxe

```
dado = getcl2C()
dado = getcl2C1()
dado = getcl2C2()
dado = ReadI2C()
dado = ReadI2C1()
dado = ReadI2C2()
```

Sendo:

- **dado:** valor do tipo unsigned char.

Exemplo

```
while (!DataRdyI2C()); //Aguarda até que o registro SSPBUF receba um novo dado.
dado = getcl2C(); //Efetua a leitura do dado.
```

15.3.9 Recepção de String

A função **getsl2C** lê uma *string* de tamanho prefixado com o dispositivo operando no modo *Master*, e retorna '0' se todos os bytes forem recebidos ou '-1' se tiver ocorrido uma colisão no barramento.

Sintaxe

```
status = getsl2C( buffer, tamanho )
status = getsl2C1( buffer, tamanho )
status = getsl2C2( buffer, tamanho )
```

Sendo:

- **buffer:** ponteiro para a matriz de caracteres localizada na memória de dados.
- **tamanho:** número de bytes que o módulo deve receber (8bits).
- **status:** valor do tipo signed char.

Exemplo

```
unsigned char buffer[15];
getsl2C( buffer, 15); //Lê 15 bytes de dados no barramento I2C.
```

15.3.10 Status do Barramento I²C

A função **IdleI2C** verifica o status do periférico I²C e aguarda até que o barramento esteja disponível

Sintaxe

```
IdleI2C()
IdleI2C1()
IdleI2C2()
```

Exemplo

```
IdleI2C(); //Aguarda até que o barramento esteja livre.
StartI2C(); //Inicia a comunicação I2C.
```

```
IdleI2C2(); //Aguarda até que o barramento esteja livre.
StartI2C2(); //Inicia a comunicação I2C.
```

15.3.11 Status do Buffer de Recepção

A função **DataRdyI2C** verifica se há um novo byte no buffer de recepção do módulo SSP e retorna '1' se há um dado; caso contrário, retorna '0'.

Sintaxe

```
status = DataRdyI2C()
status = DataRdyI2C1()
status = DataRdyI2C2()
```

Sendo:

- **status**: valor do tipo unsigned char.

Exemplo

```
while( !DataRdyI2C() ); //Aguarda até que o registro SSPBUF receba um novo dado.
dado = ReadI2C(); //Efetua a leitura do dado.
```

```
while( !DataRdyI2C1() ); //Aguarda até que o registro SSPBUF1 receba um novo dado.
dado = ReadI2C1(); //Efetua a leitura do dado.
```

15.3.12 Transmissão de Caractere

A função **WriteI2C** ou **putcI2C** envia um único byte para o barramento I²C e retorna '0' para indicar sucesso na transmissão ou '-1' para indicar que houve uma colisão de escrita.

Sintaxe

```
status = WriteI2C( dado )
status = WriteI2C1( dado )
status = WriteI2C2( dado )
status = putcI2C( dado )
status = putcI2C1( dado )
status = putcI2C2( dado )
```

Sendo:

- **dado**: valor do tipo **unsigned char**.
- **status**: valor do tipo **signed char**.

Exemplo

```
Writel2C ( dado ); //Envia um byte através da I2C do módulo SSP.
Writel2C1 ( dado ); //Envia um byte através da I2C do módulo SSP 1.
putl2C2 ( dado ); //Envia um byte através da I2C do módulo SSP 2.
```

15.3.13 Transmissão de String

A função **putsI2C** envia uma *string* de dados para o barramento I²C até que seja encontrado um caractere nulo. Ela pode operar tanto no modo *Master* como no modo *Slave*, porém a interpretação do valor devolvido pela função é diferente.

Sintaxe

```
status = putsI2C ( string )
status = putsI2C1 ( string )
status = putsI2C2 ( string )
```

Sendo:

- **string**: ponteiro para a matriz de caracteres localizada na memória de dados.
- **status**: valor do tipo **signed char**. Veja a Tabela 15.3.

Tabela 15.3: Descrição do valor retornado pelo comando de transmissão de *string*.

Modo	Valor	Descrição
Master	0	Indica que a <i>string</i> foi enviada corretamente.
	-2	Indica que o dispositivo <i>Slave</i> respondeu com um NOT ACK.
	-3	Indica a ocorrência de colisão de escrita.
Slave	0	Indica que a <i>string</i> foi enviada corretamente.
	-2	Indica que o dispositivo <i>Master</i> respondeu com um NOT ACK após o fim da transmissão.

Exemplo

```
unsigned char mensagem[ ] = "TESTE I2C";
putsI2C (mensagem); //Envia a mensagem "TESTE I2C" através da comunicação I2C.
```

15.4 Funções I²C Implementadas em Software

O MPLAB® C18 também dispõe de um conjunto de funções que permite implementar uma comunicação I²C em qualquer pino I/O do microcontrolador da família PIC18. Essas funções estão implementadas na biblioteca **sw_i2c.h**.

Os pinos I/O usados pela comunicação I²C implementada em software são, por definição, RB4 - SDA e RB3 - SCL. No entanto, eles podem ser redefinidos alterando as macros declaradas na biblioteca **sw_i2c.h**, localizada em **C:\MCC18\h**.

Tabela 15.4: Macros para a comunicação I²C.

Linha I2C	Macros	Valor padrão	Uso
Pino de DADO	DATA_PIN	PORTBbits.RB4	Pino usado para a linha de dado
	DATA_LAT	LATBbits.RB4	Latch associado com o pino de dado
	DATA_LOW	TRISBbits.TRISB4 = 0;	Define o pino de dado como saída
	DATA_HI	TRISBbits.TRISB4 = 1;	Define o pino de dado como entrada
Pino de CLOCK	SCLK_PIN	PORTBbits.RB3	Pino usado para a linha de clock
	CLK_LAT	LATBbits.LATB3	Latch associado com o pino de clock
	CLOCK_LOW	TRISBbits.TRISB3 = 0;	Define o pino de clock como saída
	CLOCK_HI	TRISBbits.TRISB3 = 1;	Define o pino de clock como entrada

Após qualquer modificação o usuário deve recomilar as rotinas, e então incluir os arquivos atualizados no projeto. Os arquivos estão localizados em C:\MCC18\src\pmc_common\SW_I2C.

Quando as funções implementadas em software forem utilizadas, pode ser necessário substituir as funções **Delay10TCY()** por **Delay10TCYx(valor)** nos seguintes arquivos.

- swacki2c.c
 - swctki2c.c
 - swqtsi2c.c

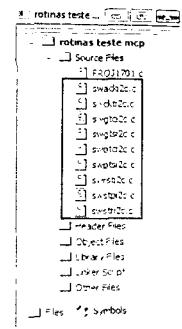


Figura 15.6 Arquivos que devem ser inseridos no projeto.



Os pinos definidos como linha de comunicação devem ser configurados como entrada.

15.4.1 Alongamento do Clock para o Modo Slave

A função **Clock_test** é usada para permitir que o dispositivo *Slave* estenda o sinal de *clock*, colocando a linha SCL em nível lógico '0', por um período de tempo prefixado. Se no final do período de atraso a linha de *clock* ainda estiver em nível baixo '0', a função retorna '-2' informando a ocorrência de um erro; caso contrário, ela retorna '0'. O tempo de atraso pode ser modificado, caso seja necessário.

Sintaxe

```
status = Clock test( )
```

Sendo:

- status: valor do tipo signed char.

Exemplo

Clock test(); //Força a linha de clock para o nível baixo '0' por um período de tempo predeterminado.

15.4.2 Condição de Acknowledge (ACK)

A função **SWAckI2C** gera uma condição de Acknowledge e retorna '0' se o Slave reconheceu o comando/dado; caso contrário, retorna '1'.

Sintaxe

resposta = **SWAckI2C** ()

Sendo:

- **resposta**: valor do tipo **unsigned char**.

15.4.3 Condição de Not Acknowledge (Not ACK)

A função **SWNotAckI2C** gera uma condição de não reconhecimento (Not Acknowledge) no barramento I²C, e retorna '0' se o Slave reconheceu o comando/dado; caso contrário, retorna '1'.

Sintaxe

resposta = **SWNotAckI2C** ()

Sendo:

- **resposta**: valor do tipo **unsigned char**.

15.4.4 Condição de RESTART

A função **SWRestartI2C** gera uma condição de RESTART no barramento I²C.

Sintaxe

SWRestartI2C ()

Exemplo

SWRestartI2C (); //Gera uma condição de RESTART.

15.4.5 Condição de START

A função **SWStartI2C** gera uma condição de START no barramento I²C.

Sintaxe

SWStartI2C ()

Exemplo

SWStartI2C (); //Gera uma condição de START.

15.4.6 Condição de STOP

A função **SWStopI2C** gera uma condição de STOP no barramento I²C.

Sintaxe

SWStopI2C ()

Exemplo

```
SWStopI2C( ); //Gera uma condição de STOP.
```

15.4.7 Recepção de Caractere

A função **SWReadI2C** ou **SWGGetI2C** lê um único byte vindo do barramento I²C.

Sintaxe

```
dado = SWGetI2C( )
dado = SWReadI2C( )
```

Sendo:

- **dado**: valor do tipo **unsigned char**.

Exemplo

```
unsigned char dado_recebido;
dado_recebido = SWGetI2C( );
```

15.4.8 Recepção de String

A função **SWGGetsI2C** lê uma *string* de tamanho prefixada no barramento I²C e retorna '0' se todos os *bytes* forem recebidos ou '-1' se o dispositivo *Master* gerar uma condição de não reconhecimento (*Not Acknowledge*), antes de todos os *bytes* serem recebidos.

Sintaxe

```
status = SWGetsI2C( buffer, tamanho )
```

Sendo:

- **buffer**: ponteiro para a matriz de caracteres localizada na memória de dados.
- **tamanho**: número de *bytes* que o módulo deve receber (8bits).
- **status**: valor do tipo **signed char**.

Exemplo

```
char buffer[20];
SWGGetsI2C( buffer,8 );
```

15.4.9 Transmissão de Caractere

A função **SWWriteI2C** ou **SWPutI2C** envia um único byte para o barramento I²C e retorna '0' para indicar sucesso na transmissão ou '-1' para indicar que houve um erro na transmissão.

Sintaxe

```
status = SWWriteI2C( dado )
status = SWPutI2C( dado )
```

Sendo:

- **dado**: valor do tipo **unsigned char**.
- **status**: valor do tipo **signed char**.

Exemplo

```
if(SWWriteI2C(0x80))
{
//Ocorreu uma colisão no barramento I2C;
}
```

15.4.10 Transmissão de String

A função **SWPutsl2C** envia uma *string* de dados para o barramento I²C até que seja encontrado um caractere nulo e retorna '0' para indicar que à *string* foi enviada corretamente ou '-1' para indicar que houve um erro de escrita no barramento I²C.

Sintaxe

```
status = SWPutsl2C ( string )
```

Sendo:

- **string**: ponteiro para a matriz de caracteres localizada na memória de dados.
- **status**: valor do tipo **signed char**.

Exemplo

```
unsigned char buffer[ ] = "TESTE I2C SOFTWARE";
SWPutsl2C (buffer); //Envia a mensagem "TESTE I2C SOFTWARE" através da comunicação I2C.
```

15.5 Projeto

O projeto proposto neste capítulo trata-se do controle de acesso de uma memória EEPROM externa através do protocolo I²C.

15.5.1 Memória EEPROM 24C128

Uma maneira simples e barata de ampliar a capacidade de armazenar dados não voláteis é empregar memórias EEPROM externas, cuja comunicação é normalmente realizada pelo protocolo I²C ou SPI (o SPI será comentado no próximo capítulo). No projeto proposto pelo livro, utilizaremos uma memória EEPROM externa conectada a um barramento I²C. A Figura 15.7 mostra os pinos do modelo 24C128 de 128K (16.384 palavras de 8bits).

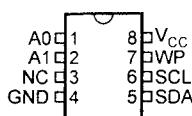


Figura 15.7: Memória 24C128.

Tabela 15.5: Descrição dos pinos.

Pino	Nome	Descrição
1	A0	Endereço do dispositivo.
2	A1	Endereço do dispositivo.
3	NC	Não conectar
4	GND	Terra.
5	SDA	Dado serial.
6	SCL	Entrada de clock.
7	WP	Proteção contra escrita. 1 - Habilitada. 0 - Desabilitada.
8	VCC	Alimentação de 2.7 a 5.5V.

15.5.2 Modo de Funcionamento

O endereço da memória EEPROM 24C128 é composto por 7 bits, conforme mostra a Figura 15.8. Os 5 bits mais significativos possuem valores fixos e somente o bit A_0 e A_1 são variáveis, limitando a quantidade dessas memórias para 4 ($2^2 = 4$) em um mesmo barramento. Porém, o dado enviado pelo microcontrolador é constituído de 8 bits; 7 bits determinam o endereço do dispositivo e 1 bit (R/W) informa o tipo de operação que será realizado (0 - escrita e 1 - leitura).

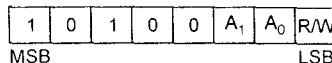


Figura 15.8: Endereço da memória 24C128.

Até o presente momento sabemos como endereçar o dispositivo, bem como selecionar o tipo de operação que será realizada. Agora vamos aprender como se fazem a escrita e a leitura de dados nessa memória.

Uma vez endereçada, ela retorna um bit de reconhecimento ($ACK = 0$), informando ao dispositivo Master que o dispositivo Slave solicitado respondeu à chamada e está pronto para receber a posição da memória em que o dado será escrito ou lido. Essa posição é composta de 16 bits e o protocolo envia somente 8 bits de dados, então é preciso segmentá-lo e enviar o byte mais significativo, seguido do menos significativo, efetuando a leitura do bit de reconhecimento (ACK) a cada dado enviado. Daqui em diante, há dois modos distintos de operação: escrita ($R/W = 0$) e leitura ($R/W = 1$) de dados.

Se for uma operação de escrita, basta enviar o dado de 8 bits e ler o bit de reconhecimento ($ACK = 0$), como forma de verificar se o byte foi corretamente recebido pelo Slave. Se for uma operação de leitura, o Master terá de enviar um comando de RESTART seguido do endereço do dispositivo, com a opção de leitura selecionada. Se o dispositivo Slave responder com $ACK = 0$, basta ler o dado (8 bits) enviado pelo Slave, o qual corresponde ao valor do dado armazenado na posição especificada da memória EEPROM externa.

Todo início e fim de uma operação de escrita ou leitura de dados deve começar com a condição de START e finalizar com STOP.

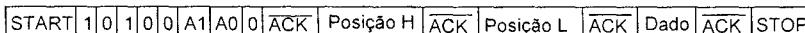


Figura 15.9: Operação de escrita.

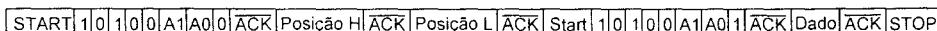


Figura 15.10. Operação de leitura.

15.5.3 Circuito Eletrônico Proposto para o Projeto

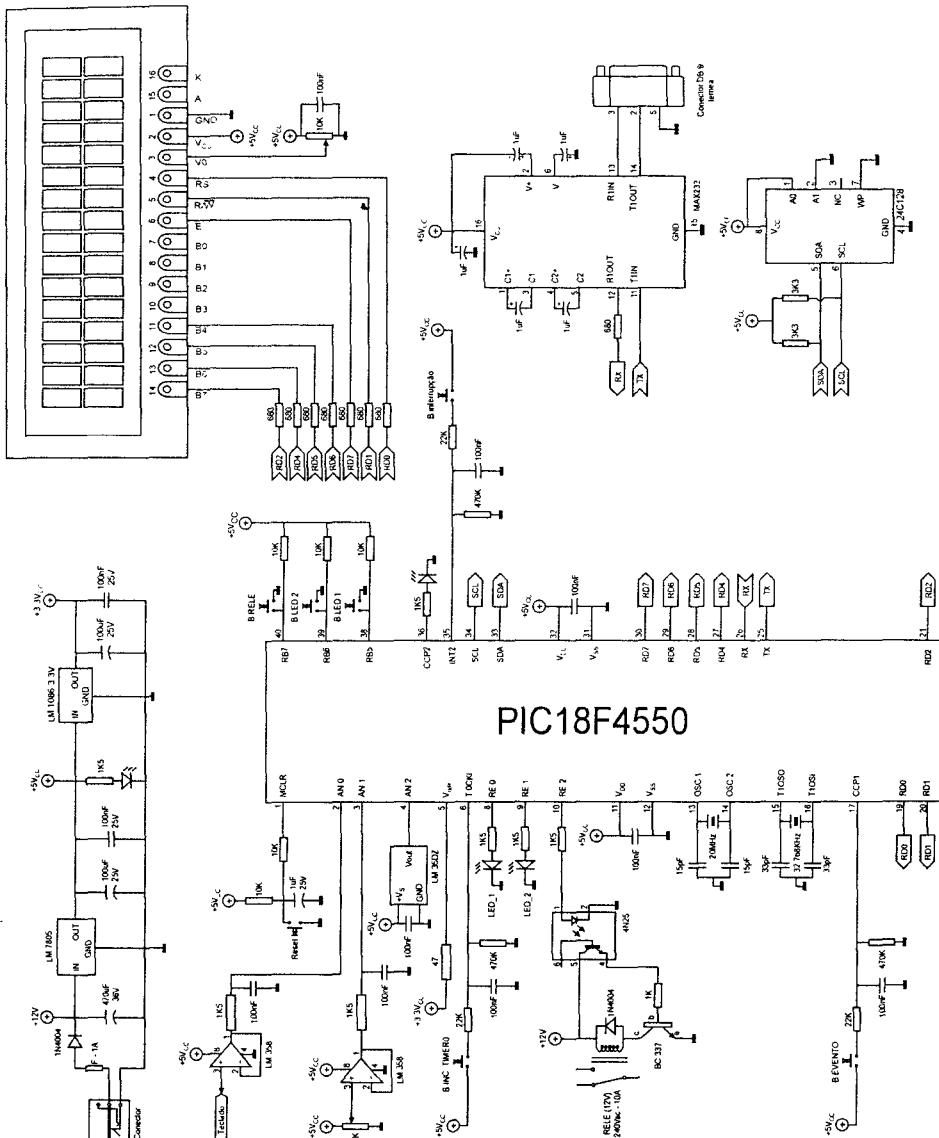


Figura 15.11: Circuito eletrônico de comunicação entre o PIC e o CI 24C128.

Código

```
*****Projeto Capítulo 15 - EEPROM I2C*****
**
** Este programa imprime no display LCD 2x16, os dados contidos em determinadas posições da memória 24Cxx.
**
** Autor: Alberto Noboru Miyadaira
*****
```

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <delays.h> //Adiciona a biblioteca de funções de delay.
#include <i2c.h> //Adiciona a biblioteca de funções de controle e configuração da comunicação I2C.
#include "C:\pic18\ biblioteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)
```

```
//Esta função escreve um dado em uma posição da memória especificada.
//Retorna o valor 0 caso não tenha ocorrido erro.
//Retorna o valor 1 caso tenha ocorrido algum erro.
unsigned char escreve_eeprom_i2c(unsigned char num_eeprom, unsigned int pos_memoria,unsigned char dado)
{
    num_eeprom = 0b10100000|(num_eeprom<<1); //Formato do dado: 1.0.1.0.0.A1.A0.0.

    IdleI2C(); //Aguarda até que o barramento esteja livre.
    StartI2C(); //Inicia a comunicação I2C.
    while ( SSPCON2bits.SEN ); //Aguarda até que a condição de START finalize.
    IdleI2C(); //Aguarda até que o barramento esteja livre.

    if (!WriteI2C(num_eeprom))//Envia o endereço da memória EEPROM
    {
        IdleI2C(); //Aguarda até que o barramento esteja livre.
        if (!WriteI2C(pos_memoria>>8))//Envia o byte mais significativo, referente à posição da memória.
        {
            IdleI2C(); //Aguarda até que o barramento esteja livre.
            if (!WriteI2C(pos_memoria))//Envia o byte menos significativo, referente à posição da memória.
            {
                IdleI2C(); //Aguarda até que o barramento esteja livre.
                if (!WriteI2C(dado))//Envia o dado que deve ser escrito na posição indicada.
                {
                    IdleI2C(); //Aguarda até que o barramento esteja livre.
                    StopI2C(); //Finaliza a comunicação I2C.
                    while ( SSPCON2bits.PEN ); //Aguarda até que a condição de STOP finalize.
                    Delay10KTCYx(5); //Gera um delay de 10 milissegundos. 5*10.000*0.2us= 10ms
                    return 0; //Comando finalizado com sucesso.
                }
            }
        }
    }
}
return 1;//Erro no comando.
```

```
}
```

```
//Esta função lê um dado em uma posição da memória especificada.
//Retorna o valor correspondente ao dado.
unsigned char le_eeprom_i2c(unsigned char num_eeprom, unsigned int pos_memoria)
{
    unsigned char dado_recebido;

    num_eeprom = 0b10100000|(num_eeprom<<1); //Formato do dado: 1.0.1.0.0.A1.A0.0.

    IdleI2C(); //Aguarda até que o barramento esteja livre.
    StartI2C(); //Inicia a comunicação I2C.
```

```

while ( SSPCON2bits SEN ); //Aguarda até que a condição de START finalize
IdleI2C( ); //Aguarda até que o barramento esteja livre.

if (!Writel2C(num_eeprom))//Envia o endereço da memória EEPROM.
{
    IdleI2C( ); //Aguarda até que o barramento esteja livre.
    if (!Writel2C(pos_memoria>8))//Envia o byte mais significativo, referente à posição da memória.
    {
        IdleI2C( ); //Aguarda até que o barramento esteja livre.
        if (!Writel2C(pos_memoria))//Envia o byte menos significativo, referente à posição da memória.
        {
            IdleI2C( ); //Aguarda até que o barramento esteja livre.
            RestartI2C( );//Envia um comando de RESTART.
            while ( SSPCON2bits.SEN ); //Aguarda até que a condição de RESTART finalize.

            IdleI2C( ); //Aguarda até que o barramento esteja livre.
            if (!Writel2C(num_eeprom|0b00000001))//Envia o endereço da memória EEPROM.
            {
                IdleI2C( ); //Aguarda até que o barramento esteja livre.
                dado_recebido = getI2C();//Lê o dado presente na posição indicada.

                IdleI2C( ); //Aguarda até que o barramento esteja livre.
                StopI2C( );//Finaliza a comunicação I2C.
                while ( SSPCON2bits.PEN ); //Aguarda até que a condição de STOP finalize.
                return dado_recebido;//Retorna o dado recebido.
            }
        }
    }
}

void main (void)
{
    TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
    TRISB = 0b11100111; //RB0, RB1, RB2, RB5, RB6 e RB7 – entrada e RB3 a RB4 – saída.
    TRISC = 0b10111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
    TRISD = 0b00000000; //RD0 a RD7 – saída.
    TRISE = 0b00000000; //RE0 a RE2 – saída.

    OpenI2C (MASTER,SLEW_OFF); //Modo Master, com velocidade de 100KHz.
    SSPADD=49; //Taxa de comunicação 100KHz.

    PORTD = 0x00, //Coloca a porta D em 0V.
    PORTE = 0x00; //Coloca a porta E em 0V.

    lcd_inicia(0x28, 0x0F, 0x06); //Inicializa o display LCD alfanumérico com quatro linhas de dados.
    lcd_LD_cursor (0); //Desliga o cursor.

    lcd_posicao (1,1); //Posiciona o cursor na L=1 e C=1;
    imprime_string_lcd ("EEPROM - 24C128"); //Imprime a string.

    //Insere dados nas posições especificadas.
    escreve_eeprom_i2c(1, 0, 'P');
    escreve_eeprom_i2c(1, 1, 'I');
    escreve_eeprom_i2c(1, 2, 'C');
    escreve_eeprom_i2c(1, 8000, 'T');
    escreve_eeprom_i2c(1, 8001, 'E');
    escreve_eeprom_i2c(1, 16381, 'S');
    escreve_eeprom_i2c(1, 16382, 'T');
}

```

```
escreve_eeprom_i2c(1, 16383, 'E');

lcd_posicao (2,1); //Posiciona o cursor na L=2 e C=1.

//Imprime no display LCD 2x16, os dados contidos nas posições especificadas.
lcd_escreve_dado(le_eeprom_i2c(1,0));
lcd_escreve_dado(le_eeprom_i2c(1,1));
lcd_escreve_dado(le_eeprom_i2c(1,2));
lcd_escreve_dado(le_eeprom_i2c(1,8000));
lcd_escreve_dado(le_eeprom_i2c(1,8001));
lcd_escreve_dado(le_eeprom_i2c(1,16381));
lcd_escreve_dado(le_eeprom_i2c(1,16382));
lcd_escreve_dado(le_eeprom_i2c(1,16383));

while (1); //Looping infinito.
}
```



Este projeto também pode ser executado utilizando as funções implementadas em software. O código alternativo para o projeto está localizado no arquivo disponibilizado pelo autor (PROJ15002.c) no site da Editora Érica (www.editoraerica.com.br).

Comunicação SPI

A interface SPI (*Serial Peripheral Interface*) foi desenvolvida pela Motorola. Trata-se de uma comunicação síncrona que opera no modo *full-duplex*, suporta um dispositivo *Master* e um ou mais *Slaves* conectados no barramento. Ela se caracteriza pela simplicidade e eficiência, podendo alcançar velocidades de até 70MHz (o microcontrolador PIC18F4550 pode alcançar 10MHz com Fosc = 40MHz).

Atualmente está presente em inúmeros dispositivos, devido à capacidade de fluxo de dados e à facilidade de implementá-lo. Dentre eles temos microcontroladores, conversores D/A e A/D, sensores de temperatura, SD/MMC Card, LCDs, potenciómetros digitais etc.

O barramento é composto de quatro linhas de comunicação denominadas de: SCLK/SCK (*Serial Clock*), MOSI/SIMO/SDI/DI/SI (*Master Output - Slave Input*), MISO/SOMI/SDO/DO/SO (*Master Input - Slave Output*) e SS/CS/STE (*Slave Select*).

Existem basicamente dois modos de fazer a conexão em um barramento SPI. Podemos conectar o *Master* e o *Slave* de modo independente, ou então conectá-los em cascata, como pode ser observado nas Figuras 16.1 e 16.2.

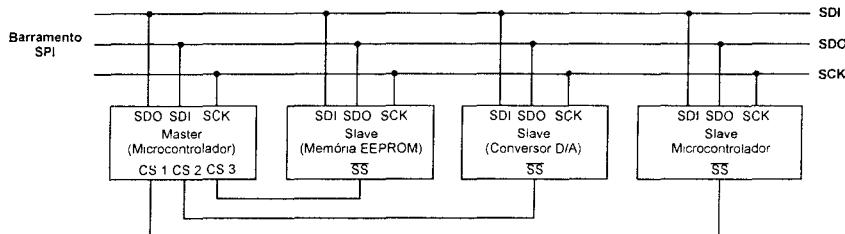


Figura 16.1: Conexão independente.

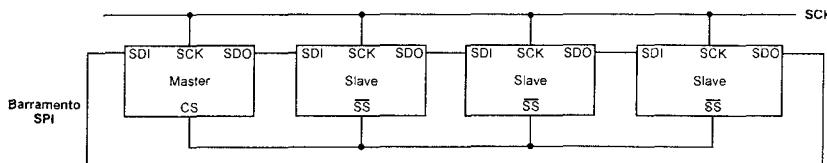


Figura 16.2: Conexão em cascata.

Uma grande parcela de dispositivos *Slaves* possui a saída *tri-state*, ou seja, quando não estão selecionados, as suas saídas assumem o estado de alta impedância. Os dispositivos que não possuem a saída *tri-state* não devem compartilhar os mesmos segmentos do barramento SPI com outros *Slaves*.

16.1 Funcionamento do Protocolo SPI

Na comunicação SPI, os dispositivos configurados como *Slave* são selecionados pelo pino SS, em vez de possuírem um endereço, como é visto na interface I²C. Esta característica, associada à capacidade de operar no modo *full-duplex*, permite que a interface SPI alcance taxas de comunicação superiores à interface I²C. Contudo, devido à falta de padronização, a seleção do dispositivo *Slave* pode ser feita tanto em nível lógico alto '1' quanto em nível lógico baixo '0', sendo esta última a mais comum.

Os dados transferidos são normalmente representados por 8bits, cuja transmissão pode ser iniciada pelo bit menos significativo (LSb) ou mais significativo (MSb), com a possibilidade de variar conforme o dispositivo.

O sinal de *clock* desse protocolo pode ser configurado de quatro formas distintas, as quais resultam da combinação entre a polaridade (CPOL) e a fase (CPHA) do *clock*, como pode ser visto na Tabela 16.1.

Tabela 16.1: Configuração do *clock*.

Modo	CPOL (CKP)	CPHA (CKE)	Descrição
0	0	0	A polaridade do <i>clock</i> é 0 e o pulso de <i>clock</i> é gerado no começo do período.
1	0	1	A polaridade do <i>clock</i> é 0 e o pulso de <i>clock</i> é gerado no final do período.
2	1	0	A polaridade do <i>clock</i> é 1 e o pulso de <i>clock</i> é gerado no começo do período.
3	1	1	A polaridade do <i>clock</i> é 1 e o pulso de <i>clock</i> é gerado no final do período.

Veja na Figura 16.3 o formato do sinal de *clock* de acordo com o modo de operação selecionado.

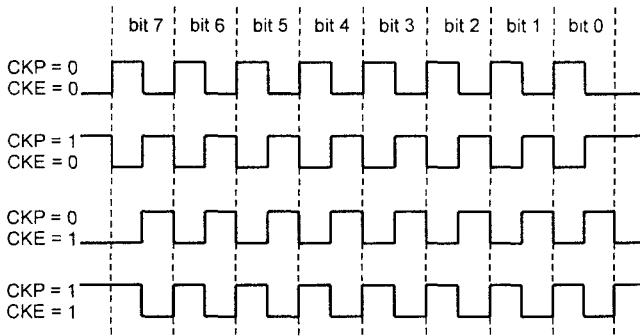


Figura 16.3: Forma de onda do modo SPI (Master).

Uma vez que as configurações iniciais foram realizadas, os dispositivos conectados ao barramento estão prontos para se comunicarem.

A comunicação entre dois dispositivos é iniciada a partir do momento em que o *Master* seleciona o *Slave* com o qual deseja estabelecer uma comunicação, colocando o pino SS do dispositivo *Slave* no estado *low* '0'. Deste ponto em diante, o *Slave* se prepara para se comunicar com o *Master*, no entanto alguns dispositivos não são capazes de responder de imediato, necessitando de um pequeno atraso, antes de receberem o

sinal de *clock*. Ao mesmo tempo em que o *Master* envia um dado para o *Slave*, ele também recebe um dado.

Após realizar as operações desejadas, o *Master* finaliza a comunicação, interrompendo o envio do sinal de *clock* e desabilitando o dispositivo *Slave* ($\overline{SS} = 1$).

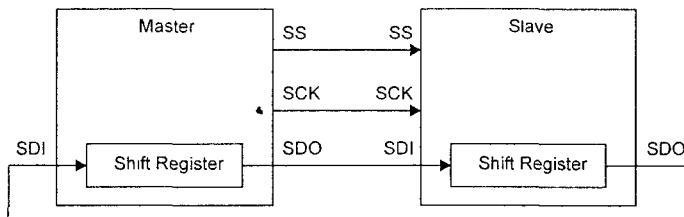


Figura 16.4: Conexão Master-Slave.



Lembre-se de que nem todos os dispositivos são selecionados no estado *low* ($\overline{SS} = 0$). Alguns são selecionados no estado *high* ($SS = 1$), embora sejam incomuns.

16.2 Comunicação SPI do PIC18F4550

A interface SPI, nos microcontroladores da família PIC18, está implementada no módulo MSSP (*Master Synchronous Serial Port*) ou apenas SSP (*Synchronous Serial Port*), ambas capazes de operar no modo SPI ou I²C. Como as duas interfaces compartilham um mesmo módulo, somente uma delas pode estar ativa por vez.

O microcontrolador PIC18F4550 possui uma interface SPI, cuja taxa de comunicação pode alcançar velocidade de 10Mbps, com Fosc = 40MHz. Quando configurado como *Slave*, é ativo em *low* '0' e a saída SDO assume o estado de alta impedância, caso o dispositivo esteja inativo.

Os pinos pertencentes ao periférico SPI são:

- » **SCK**: pino de entrada ou saída do sinal de *clock*.
- » **SDI**: pino de entrada de dado.
- » **SDO**: pino de saída de dado.
- » **SS** : pino de seleção (ativo em nível '0'). Usado no modo *Slave*.



O sentido de fluxo dos pinos deve ser definido pelo registro **TRIS**. Se o dispositivo estiver configurado como *Master*, o pino SCK deve ser definido como saída (**TRISB<1>** = 0); caso contrário, deve ser configurado como entrada (**TRISB<1>** = 1).

16.3 Funções de Controle/Configuração do Periférico SPI

As funções de controle e configuração do periférico SPI dos microcontroladores da família PIC18 estão declaradas na biblioteca **spi.h**. As funções apresentadas na sequência possuem três formatos. Os nomes das funções, cujas últimas letras são '1' ou '2', devem ser utilizadas em dispositivos dotados de mais de um periférico SPI; caso contrário, se não apresentar nenhum número associado à última letra, deve ser usada em dispositivos que contenham apenas um periférico SPI.



16.3.1 Configura o Periférico SPI

A função **OpenSPI** configura e habilita o periférico SPI.

Sintaxe

OpenSPI (modo, modo_clock, f_amostra)

OpenSPI1 (modo, modo_clock, f_amostra)

OpenSPI2 (modo, modo_clock, f_amostra)

Sendo:

- **modo**: uma das constantes listadas na Tabela 16.2.
- **modo_clock**: uma das constantes listadas na Tabela 16.3.
- **f_amostra**: uma das constantes listadas na Tabela 16.4.

Tabela 16.2: Define o modo de operação da interface SPI.

Constante	Descrição
SPI_FOSC_4	Modo Master, Sinal de <i>clock</i> = Fosc / 4.
SPI_FOSC_16	Modo Master, Sinal de <i>clock</i> = Fosc / 16.
SPI_FOSC_64	Modo Master, Sinal de <i>clock</i> = Fosc / 64.
SPI_FOSC_TMR2	Modo Master, Sinal de <i>clock</i> = Saída do TMR2 / 2.
SLV_SSON	Modo Slave. Pino de controle <i>SS</i> habilitado.
SLV_SSOFF	Modo Slave. Pino de controle <i>SS</i> desabilitado.

Tabela 16.3: Define o modo de operação do sinal de *clock*.

Constante	Descrição
MODE_00	A polaridade do <i>clock</i> é 0 e o pulso de <i>clock</i> é gerado no começo do período.
MODE_01	A polaridade do <i>clock</i> é 0 e o pulso de <i>clock</i> é gerado no final do período.
MODE_10	A polaridade do <i>clock</i> é 1 e o pulso de <i>clock</i> é gerado no começo do período.
MODE_11	A polaridade do <i>clock</i> é 1 e o pulso de <i>clock</i> é gerado no final do período.

Tabela 16.4: Define a fase de leitura do dado.

Constante	Descrição
SMPEND	Lê o dado de entrada no final do período.
SMPMID	Lê o dado de entrada na metade do período.

Exemplo

OpenSPI (SPI_FOSC_4, MODE_01, SMPMID); //Configura o periférico SPI do módulo SSP.

OpenSPI1(SPI_FOSC_64, MODE_11, SMPEND); //Configura o periférico SPI do módulo SSP 1.

OpenSPI2(SPI_FOSC_16, MODE_10, SMPMID); //Configura o periférico SPI do módulo SSP 2.

16.3.2 Desabilita o Periférico SPI

A função **CloseSPI** desabilita o periférico SPI.

Sintaxe

```
CloseSPI( )
CloseSPI1( )
CloseSPI2( )
```

Exemplo

```
CloseSPI( );           // Desabilita o periférico SPI do módulo SSP.
CloseSPI1( );         // Desabilita o periférico SPI do módulo SSP 1.
CloseSPI2( );         // Desabilita o periférico SPI do módulo SSP 2.
```

16.3.3 Recepção de Caractere

A função **ReadSPI** ou **getcSPI** retorna um *byte* de dado durante o ciclo de leitura.

Sintaxe

```
dado = getcSPI( )
dado = getcSPI1( )
dado = getcSPI2( )
dado = ReadSPI( )
dado = ReadSPI1( )
dado = ReadSPI2( )
```

Sendo:

- **dado**: valor do tipo **unsigned char**.

Exemplo

```
If(DataRdySPI( )); //Verifica se há um novo dado no registro SSPBUF.
                    dado = getcSPI( ); //Efetua a leitura do dado.
```

16.3.4 Recepção de String

A função **getsSPI** lê uma *string* de tamanho prefixado.

Sintaxe

```
getsSPI( buffer, tamanho )
getsSPI1( buffer, tamanho )
getsSPI2( buffer, tamanho )
```

Sendo:

- **buffer**: ponteiro para a matriz de caracteres localizada na memória de dados.
- **tamanho**: número de *bytes* que o módulo deve receber (*8bits*).

Exemplo

```
unsigned char buffer[15];
getsSPI( buffer, 15 ); //Lê 15 bytes de dados no barramento SPI.
```

16.3.5 Status do Buffer de Recepção

A função **DataRdySPI** verifica se há um novo *byte* no *buffer* de recepção do módulo SSP, e retorna '1' se há um dado; caso contrário, retorna '0'.

Sintaxe

```
status = DataRdySPI( )
status = DataRdySPI1( )
status = DataRdySPI2( )
```

Sendo:

- **status:** valor do tipo **unsigned char**.

Exemplo

```
while ( !DataRdySPI( ) ); //Aguarda até que o registro SSPBUF receba um novo dado.
dado = getcSPI( ); //Efetua a leitura do dado.
```

```
while ( !DataRdySPI2( ) ); //Aguarda até que o registro SSPBUF2 receba um novo dado.
dado = getcSPI2( ); //Efetua a leitura do dado.
```

16.3.6 Transmissão de Caractere

A função **WriteSPI** ou **putcSPI** envia um único *byte* para o barramento SPI, e retorna '0' para indicar sucesso na transmissão ou '-1' para indicar que houve uma colisão de escrita.

Sintaxe

```
status = WriteSPI( dado )
status = WriteSPI1( dado )
status = WriteSPI2( dado )
status = putcSPI( dado )
status = putcSPI1( dado )
status = putcSPI2( dado )
```

Sendo:

- **dado:** valor do tipo **unsigned char**.
- **status:** valor do tipo **signed char**.

Exemplo

```
WriteSPI( dado ); //Envia um byte através da SPI do módulo SSP.
putcSPI1( dado ); //Envia um byte através da SPI do módulo SSP 1.
```

```
if(WriteSPI2( dado )) //Envia um byte através da SPI do módulo SSP 2 e verifica a ocorrência de erro.
{ //Erro na transmissão. }
```

16.3.7 Transmissão de String

A função **putsSPI** envia uma *string* de dados para o barramento SPI até que seja encontrado um caractere nulo.

Sintaxe

```
status = putsSPI( string )
status = putsSPI1( string )
status = putsSPI2( string )
```

Sendo:

- **string:** ponteiro para a matriz de caracteres localizada na memória de dados.

Exemplo

```
unsigned char mensagem[ ] = "TESTE SPI";
putsSPI (mensagem); //Envia a mensagem "TESTE SPI" através da comunicação SPI.
```

16.4 Funções SPI Implementadas em Software

O MPLAB® C18 também dispõe de um conjunto de funções que permite implementar uma comunicação SPI em qualquer pino I/O do microcontrolador da família PIC18. Essas funções estão implementadas na biblioteca **sw_spi.h**.

Os pinos I/O usados pela comunicação SPI implementada em software são, por definição, RB2 - \overline{CS} , RB3 - DI, RB7 - DO e RB6 - SCK, e podem ser redefinidos, alterando as macros declaradas na biblioteca **sw_spi.h**, localizada em C:\MCC18\h.

Tabela 16.5: Macros para a comunicação SPI.

Linha SPI	Macros	Valor padrão	Uso
Pino de seleção (\overline{CS})	SW_CS_PIN	PORTRBbits.RB2	Pino usado para seleção do chip.
	TRIS_SW_CS_PIN	TRISRBbits.TRISB2	Bit de controle da direção do pino associado com a linha \overline{CS} .
Pino de entrada de dado (DI)	SW_DIN_PIN	PORTRBbits.RB3	Pino usado como entrada de dado
	TRIS_SW_DIN_PIN	TRISRBbits.TRISB3	Bit de controle da direção do pino associado com a linha DI.
Pino de saída de dado (DO)	SW_DOUT_PIN	PORTRBbits.RB7	Pino usado como saída de dado.
	TRIS_SW_DOUT_PIN	TRISRBbits.TRISB7	Bit de controle da direção do pino associado com a linha DO.
Pino de clock (SCK)	SW_SCK_PIN	PORTRBbits.RB6	Pino usado como linha de clock
	TRIS_SW_SCK_PIN	TRISRBbits.TRISB6	Bit de controle da direção do pino associado com a linha SCK.

Além dos pinos, também é possível configurar um dos quatro modos de operação, bastando definir uma das macros listadas na Tabela 16.6.

Após qualquer modificação, o usuário deve recompilar as rotinas, e então incluir os arquivos atualizados no projeto. Os arquivos estão localizados em C:\MCC18\src\pmc_common\SW_SPI, Figura 16.5.

Tabela 16.6: Macros para a seleção do modo de operação.

Macros	Valor padrão	Descrição
MODE0	Definido	CKP = 0 CKE = 0
MODE1	Não está definido	CKP = 1 CKE = 0
MODE2	Não está definido	CKP = 0 CKE = 1
MODE3	Não está definido	CKP = 1 CKE = 1

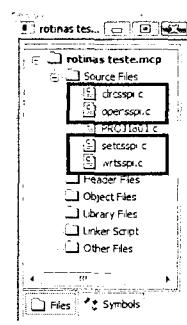


Figura 16.5: Arquivos que devem ser inseridos no projeto.

16.4.1 Configura os Pinos I/O

A função **OpenSWSPI** configura os pinos I/O para serem usados como comunicação SPI, implementada em *software*.

Sintaxe

OpenSWSPI()

Exemplo

```
OpenSWSPI(); //Configura os pinos I/O.
```

16.4.2 Transmissão de Caracteres

A função **WriteSWSPI** ou **putcSWSPI** envia um único byte para o barramento SPI implementado em *software*, e retorna o byte de dado que foi lido. Essas funções podem ser usadas tanto para leitura quanto para escrita de dados.

Sintaxe

```
dado_recebido = WriteSWSPI(dado_enviado);  
dado_recebido = putcSWSPI(dado_enviado);
```

Sendo:

- **dado_enviado**: valor do tipo **unsigned char**.
- **dado_recebido**: valor do tipo **unsigned char**.

Exemplo

```
char end = 0x10;  
char resposta;  
resposta = putcSWSPI(end);
```

16.4.3 Limpa o Pino Chip Select (CS)

A função **ClearCSSSWSPI** limpa o pino *Chip Select* (\overline{CS}).

Sintaxe

ClearCSSSWSPI()

Exemplo

```
ClearCSSSWSPI(); //Limpa o pino Chip Select.
```

16.4.4 Seta o Pino Chip Select (CS)

A função **SetCSSSWSPI** seta o pino *Chip Select* (\overline{CS}).

Sintaxe

SetCSSSWSPI()

Exemplo

```
SetCSSSWSPI(); //Seta o pino Chip Select
```

16.5 Exemplo

Veremos neste exemplo como conectar três microcontroladores PIC18 em um mesmo barramento SPI, como pode ser observado na Figura 16.6.

O dispositivo configurado como *Master* é o microcontrolador PIC18F4550, e os PIC18F2550 e PIC18F4520 são configurados como *Slaves*.

Sempre que o dispositivo *Master* selecionar o microcontrolador PIC18F2550, ele lhe retorna os status dos LEDs (LED 1 e LED 2), os quais são alterados através de uma interrupção externa INT2. Caso o microcontrolador PIC18F4520 seja selecionado, ele retorna o valor da temperatura local.

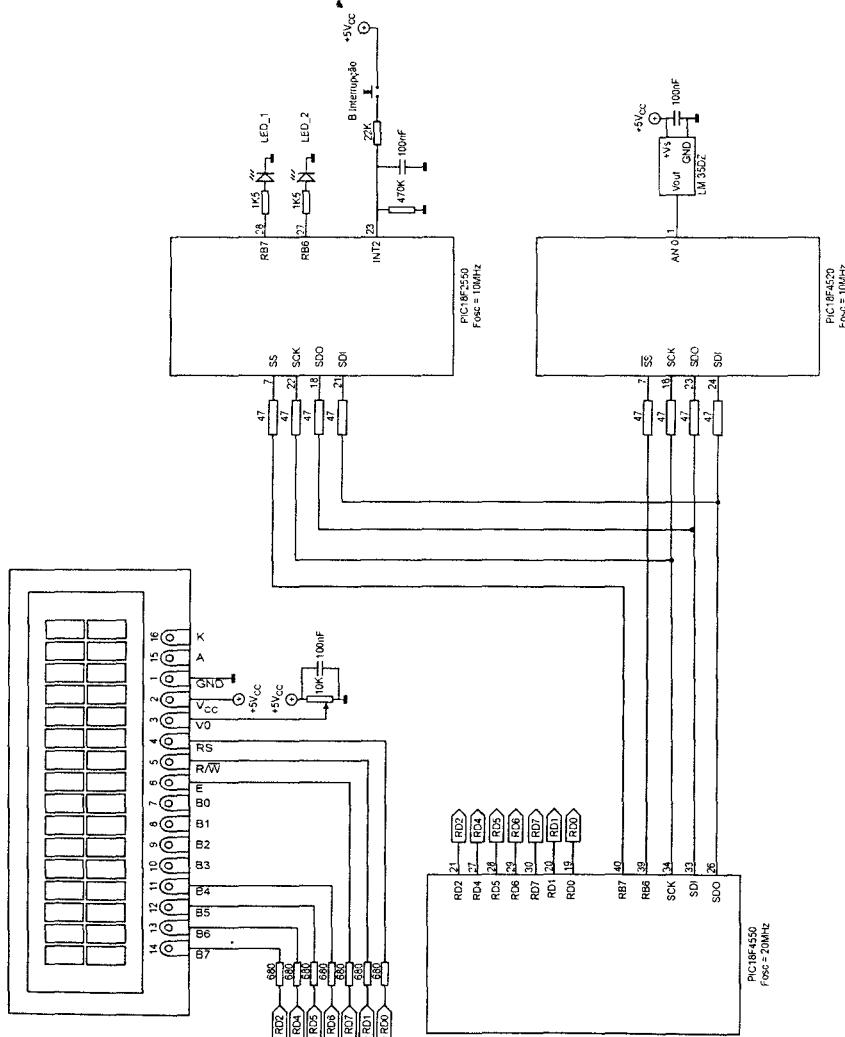


Figura 16.6: Comunicação SPI entre microcontroladores PIC.

Resistores adicionais

Código PIC18F4550 (Master)

```
***** Projeto Capítulo 16 - Dispositivo Master *****
```

```
** Sólicita aos dispositivos configurados como Slaves a enviarem informações conforme em seguida:
```

```
** Slave (PIC18F2550): Status dos LEDs.
```

```
** Slave (PIC18F4520): Temperatura ambiente.
```

```
** Autor: Alberto Noboru Miyadaira
```

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.  
#include <adc.h> //Adiciona a biblioteca de funções para o módulo conversor A/D.  
#include <delays.h> //Adiciona a biblioteca de funções de delay.  
#include <spi.h> //Adiciona a biblioteca de funções da comunicação SPI, implementada em hardware.  
#include "C:\pic18\ biblioteca_lcd_2x16.h" //Biblioteca contendo as funções do LCD.  
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)
```

```
//Variável Global.
```

```
unsigned char buffer[8];
```

```
#define SS_2550 PORTBbits.RB7 //Seleciona o MCU PIC18F2550.  
#define SS_4520 PORTBbits.RB6 //Seleciona o MCU PIC18F4520.
```

```
void main ()//Função principal.
```

```
{
```

```
TRISBbits.TRISB7 = 0; //Define o pino RB7 como saída. (Pino de seleção do PIC18F2550.)  
TRISBbits.TRISB6 = 0; //Define o pino RB6 como saída. (Pino de seleção do PIC18F4520.)
```

```
TRISBbits.TRISB1 = 0; //Define o pino SCK como saída.
```

```
TRISBbits.TRISB0 = 1; //Define o pino SDI como entrada
```

```
TRISCbits.TRISC7 = 0; //Define o pino SDO como saída.
```

```
//Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital. (Capítulo 13)
```

```
//Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
```

```
OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h.
```

```
CloseADC (); //Requer a biblioteca adc.h.
```

```
OpenSPI (SPI_FOSC_16, MODE_00 ,SMPMID);
```

```
//Configura o modo de operação do módulo SPI interno do PIC.
```

```
//Módulo configurado como master, clock = Fosc/prescaler = 20M/16 = 1,25MHz.
```

```
//Para que a interface SPI funcione adequadamente, a frequência de clock enviada pelo dispositivo Master
```

```
//deve ser menor ou igual à máxima frequência do dispositivo Slave de menor taxa de comunicação.
```

```
//CKE = 1 - CKP = 0 - SMP = 0.
```

```
SS_2550 = 1; //Desabilita o PIC18F2550.
```

```
SS_4520 = 1; //Desabilita o PIC18F4520.
```

```
lcd_inicia( 0x28, 0x0F, 0x06 ); //Iniciaiza o display LCD alfanumérico com quatro linhas de dados.
```

```
lcd_LD_cursor (0); //Desliga o cursor.
```

```
while (1) //Looping infinito.
```

```
{
```

```
SS_2550 = 0; //Habilita o PIC18F2550.
```

```
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
```

```
WriteSPI('1'); //Solicita o Status do LED 1.
```

```
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
```

```
buffer[0] = ReadSPI(); //Status do LED_1.
```

```
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
```

```

WriteSPI("2"); //Solicita o Status do LED 2.
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
buffer[1] = ReadSPI( ); //Status do LED_2.
SS_2550 = 1; //Desabilita o PIC18F2550.

SS_4520 = 0; //Habilita o PIC18F4520.
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
WriteSPI("T"); //Solicita o valor da temperatura.
Delay1KTCYx (150); //Gera um delay de 8us. T = (4*150.000)/Tosc = 30ms.
buffer[2] = ReadSPI( );
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
buffer[3] = ReadSPI( );
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
buffer[4] = ReadSPI( );
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
buffer[5] = ReadSPI( );
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
buffer[6] = ReadSPI( );
Delay10TCYx (4); //Gera um delay de 8us. T = (4*40)/Fosc = 8us.
SS_4520 = 1; //Desabilita o PIC18F4520.

lcd_limpa_tela(); // Limpa a tela do Display LCD.
lcd_posicao (1,1); // Desloca o ponteiro para a L=1 e C=1.

//Mostra os status dos LEDs do PIC18F2550.
imprime_string_lcd("LED 1."); //Imprime uma string no display.
lcd_escreve_dado(buffer[0]); //Imprime o status do LED 1.

lcd_posicao (1,9); // Desloca o ponteiro para a L=1 e C=9.
imprime_string_lcd("LED 2."); //Imprime uma string no display.
lcd_escreve_dado(buffer[1]); //Imprime o status do LED 1.

lcd_posicao (2,1); // Desloca o ponteiro para a L=2 e C=1

//Mostra o valor da temperatura medida pelo PIC18F4520
imprime_string_lcd("TEMP: "); //Imprime uma string no display.
lcd_escreve_dado(buffer[2]);
lcd_escreve_dado(buffer[3]);
lcd_escreve_dado(buffer[4]);
lcd_escreve_dado(buffer[5]);
lcd_escreve_dado(buffer[6]);
imprime_string_lcd(" C"); //Imprime uma string no display.

//Aguarda 1 segundo.
Delay10KTCYx (250); //Gera um delay de 500ms. T = (4*2.5M)/Fosc = 500.000us.
Delay10KTCYx (250); //Gera um delay de 500ms. T = (4*2.5M)/Fosc = 500.000us.

```

Código PIC18F2550 (Slave)

```
***** Projeto Capítulo 16 - Dispositivo Slave *****
**
** Envia os Status dos LEDs sempre que o dispositivo Master solicitar.
**
** Autor: Alberto Noboru Miyadaira
*****
```

```
#include <p18f2550.h> //Arquivo de cabeçalho do PIC18F2550.
#include <spi.h>      //Adiciona a biblioteca de funções da comunicação SPI, implementada em hardware.
#include <adc.h>       //Adiciona a biblioteca de funções para o módulo conversor A/D.

// Fosc = 10MHz
// Tciclo = 4/Fosc = 0.4us
#pragma config FOSC = HS
#pragma config CPUDIV = OSC1_PLL2

#pragma config WDT = OFF           //Desabilita o Watchdog Timer (WDT).
#pragma config PWRT = ON           //Habilita o Power-up Timer (PWRT).
#pragma config BOR = ON            //Brown-out Reset (BOR) habilitado somente no hardware.
#pragma config BORV = 1             //Voltagem do BOR é 4,33V.
#pragma config PBADEN = OFF         //RB0,1,2,3 e 4 configurado como I/O digital.
#pragma config LVP = OFF            //Desabilita o Low Voltage Program.

void ISR_alta_prioridade(void); //Protótipo da função de interrupção.
void ISR_baixa_prioridade(void); //Protótipo da função de interrupção.

#define LED_1 PORTBbits.RB7 //Define um nome para a estrutura.
#define LED_2 PORTBbits.RB6 //Define um nome para a estrutura.

#pragma code int_alta=0x08 //Vetor de interrupção de alta prioridade, ou padrão.
void int_alta (void)
{
    _asm GOTO ISR_alta_prioridade _endasm //Desvia o programa para a função ISR_alta_prioridade.
}
#pragma code

#pragma code int_baixa=0x18 //Vetor de interrupção de baixa prioridade, ou padrão.
void int_baixa (void)
{
    _asm GOTO ISR_baixa_prioridade _endasm //Desvia o programa para a função ISR_baixa_prioridade.
}
#pragma code

#pragma interrupt ISR_alta_prioridade
void ISR_alta_prioridade(void)
{
char temp,
    if(PIR1bits.SSPIF == 1) //Verifica a fonte de interrupção. (MSSP)
    {
        temp = SSPBUF;
        switch(temp)
        {
            case '1':
                if(LED_1 == 1)
                    SSPBUF='L';
                else
                    SSPBUF='D';
        }
    }
    while(SSPSTATbits.BF); //Aguarda até que o registro SSPBUF seja esvaziado.
}
```

```

        break;

    case '2':
        if(LED_2 == 1)
            SSPBUF='L';
        else
            SSPBUF='D';

        while(SSPSTATbits.BF); //Aguarda até que o registro SSPBUF seja esvaziado.
        break;
    }
    PIR1bits.SSPIF = 0; //Limpa o Flag bit.
}
}

#pragma interrupt ISR_baixa_prioridade
void ISR_baixa_prioridade(void)
{
    if(INTCON3bits.INT2IF == 1) //Verifica a fonte de interrupção. (INT2)
    {
        LED_1 = ~LED_1; //Inverte o Status do LED_1.
        LED_2 = ~LED_1; //Status do LED_2 é inverso do LED_1.
        INTCON3bits.INT2IF = 0; //Limpa o Flag bit.
    }
}

void main ()//Função principal.
{
    TRISBbits.TRISB7 = 0; //Define o pino RB7 como saída.
    TRISBbits.TRISB6 = 0; //Define o pino RB6 como saída.
    TRISBbits.TRISB2 = 1; //Define o pino RB2 como entrada.

    TRISBbits.TRISB1 = 1; //Define o pino SCK como entrada.
    TRISBbits.TRISB0 = 1; //Define o pino SDI como entrada.
    TRISCbits.TRISC7 = 0; //Define o pino SDO como saída.
    TRISAbits.TRISA5 = 1; //Define o pino SS como entrada.

    //Configura todas as portas multiplexadas com o módulo conversor A/D, como I/O digital. (Capítulo 13)
    //Em seguida, desabilita o conversor A/D e a interrupção associada a ele.
    OpenADC (0x00, 0x00, ADC_0ANA); //Requer a biblioteca adc.h.
    CloseADC (); //Requer a biblioteca adc.h.

    OpenSPI (SLV_SS0N, MODE_00 ,SMPMID);
    //Configura o modo de operação do módulo SPI interno do PIC.
    //Módulo configurado como Slave, clock_max = Fosc/prescaler = 10M/4 = 2.5MHz.
    //CKE = 1 - CKP = 0 - SMP = 0.

    INTCON2bits.INTEDG2 = 0 // Interrupção externa INT2 na borda de descida.
    INTCON3bits.INT2IF = 0; // Limpa o flag bit da interrupção externa INT2.
    INTCON3bits.INT2IP = 0; // Baixa prioridade.
    INTCON3bits.INT2IE = 1; // Ativa a interrupção externa INT2 (RB2).

    PIR1bits.SSPIF = 0; // Limpa o flag bit da interrupção externa INT2.
    PIR1bits.SSPIP = 1; // Alta prioridade.
    PIE1bits.SSPIE = 1; // Ativa a interrupção do módulo MSSP.

    RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo.
    INTCONbits.GIEH = 1; //Habilita todas as interrupções de alta prioridade.
    INTCONbits.GIEL = 1; //Habilita todas as interrupções de baixa prioridade.

    while (1); //Looping infinito.
}
}

```

Código PIC18F4520 (Slave)

```

***** Projeto Capítulo 16 - Dispositivo Slave *****
** Envia a temperatura ambiente.
**
** Autor: Alberto Noboru Miyadaira
***** */

#include <p18f4520.h>           //Arquivo de cabeçalho do PIC18F4520.
#include <spi.h>                 //Adiciona a biblioteca de funções da comunicação SPI, implementada em hardware.
#include <adc.h>                 //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <stdio.h>                //Adiciona a biblioteca padrão de entrada e saída.
#include <stdlib.h>               //Adiciona a biblioteca de funções miscelâneas.
#include <string.h>               //Adiciona a biblioteca de manipulação de string.

// Fosc = 10MHz
// Tocilo = 4/Fosc = 0,4us
#pragma config OSC = HS

#pragma config WDT = OFF          //Desabilita o Watchdog Timer (WDT).
#pragma config PWRT = ON           //Habilita o Power-up Timer (PWRT).
#pragma config LVP = OFF           //Desabilita o Low Voltage Program

//Variável Global.
unsigned char buffer[5];

//Converte o valor devolvido pelo conversor em um valor correspondente à tensão [mV].
float converte_tensao (float valor_conversor)
{
    return (valor_conversor*5000)/1023; //Neste caso, a tensão de entrada no pino Vref+ é 5V.
}

//Converte o valor de tensão em um valor correspondente à temperatura [C]
//Sensor de temperatura: LM35D - 10mV/C. Range de temperatura -55C a +150C
float converte_temperatura (float valor_tens)
{
    return valor_tens/10;
}

//Filtro em software.
//Obtém 256 amostras e retorna a média.
unsigned int filtro_canal()
{
    unsigned int cont_filtro;
    unsigned long valor_canal = 0;

    for( cont_filtro = 0 ; cont_filtro < 256 ; cont_filtro++ )
    {
        ConvertADC ();                      //Inicia a conversão.
        while (BusyADC ());                //Aguarda o fim da conversão.
        valor_canal += ReadADC();          //Armazena o resultado da conversão.
    }
    return (valor_canal >> 8); // Esta operação é igual a valor_canal/256.
}

void main ()//Função principal.
{
//Variáveis locais.
unsigned int valor; // Armazena o valor devolvido pelo conversor A/D.
float valor_tensao , valor_temperatura; //Armazena o valor da tensão do LM35D.
}

```

```

TRISAbits.TRISA0 = 1; //Define o pino AN0 como entrada.
TRISAbits.TRISA5 = 1; //Define o pino SS como entrada.
TRISCbits.TRISC3 = 1; //Define o pino SCK como entrada.
TRISCbits.TRISC4 = 1; //Define o pino SDI como entrada.
TRISCbits.TRISC5 = 0; //Define o pino SDO como saída.

OpenADC (ADC_FOSC_8 //Seleção da fonte de clock para a conversão A/D. Fosc = 10MHz. Tad = 0,8us
         &ADC_RIGHT_JUST      //Resultado justificado para a direita.
         &ADC_4_TAD,           //Configuração do tempo de aquisição automático. (4*Tad = 3,2us)
         ADC_CH0,              //Seleciona o canal 0 (AN0)
         &ADC_INT_OFF,          //Interrupção desabilitada.
         &ADC_VREFPLUS_VDD,    //Aref+ = Vref+
         &ADC_VREFMINUS_VSS,   //Vref- = Vss
         ADC_1ANA);            //Habilita o canal AN0.

OpenSPI (SLV_SS0N, MODE_00 ,SMPMID);
//Configura o modo de operação do módulo SPI interno do PIC.
//Módulo configurado como Slave, clock_max = Fosc/prescaler = 10M/4 = 2,5MHz.
//CKE = 1 - CKP = 0 - SMP = 0.

while (1) //Looping infinito
{
    if(ReadSPI() == 'T')
    {
        valor = filtro_canal (); //Chama a função para fazer a leitura do canal.

        //Converte o valor retornado pelo conversor A/D em um valor de tensão correspondente
        valor_tensao = converte_tensao(valor);
        valor_temperatura = converte_temperatura (valor_tensao); //Converte a tensão em temperatura
        sprintf(buffer, "%02d.%02d", (char)valor_temperatura,(char)((valor_temperatura-(char)valor_temperatura)*100));
        //Envia a string formatada para o buffer.

        //Envia o valor da temperatura através do barramento SPI.
        WriteSPI(buffer[0]);
        WriteSPI(buffer[1]);
        WriteSPI(buffer[2]);
        WriteSPI(buffer[3]);
        WriteSPI(buffer[4]);
    }
}
}

```



SD Card

É comum encontrar códigos relacionados à comunicação entre um microcontrolador e uma memória SD Card, porém pouco se fala sobre o seu modo de funcionamento. Este capítulo foi adicionado ao livro com objetivo de fazer uma breve introdução sobre a memória SD Card sem entrar em detalhes em formatação FAT. Ao terminar este capítulo, o leitor será capaz de armazenar e ler os dados presentes nessa memória, ficando a cargo do programador criar uma formatação FAT para que os dados inseridos no cartão possam ser compreendidos pelo sistema operacional WINDOWS ou qualquer outro sistema.

Memórias SD Card (*Security Data Card*) são do tipo FLASH presentes na maioria das câmeras fotográficas, celulares, *palmtops*, *datalogger*, entre outros produtos eletrônicos. Elas se popularizaram pelo baixo custo, confiabilidade, velocidade de comunicação, capacidade de armazenamento, além de possuírem dimensões reduzidas.

A principal diferença entre a memória FLASH e a EEPROM, em termos de armazenamento e acesso, é o fato de que na memória EEPROM permite acessar o dado contido em uma posição específica da memória, bastando informar o endereço em que se deseja ler ou escrever um dado de 8bits.

Na memória FLASH, os dados são organizados em blocos, e cada bloco contém um determinado número de bytes definido pelo usuário. Deste modo, se o usuário quiser acessar um determinado dado, é necessário acessar o bloco ao qual ele pertence. Veja na Figura 17.1 uma foto ilustrativa dos três formatos de memórias SD Card disponíveis no mercado.

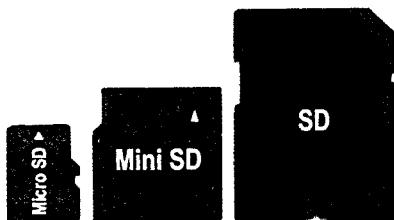


Figura 17.1: Memórias SD Card.

A comunicação entre o microcontrolador e a memória SD Card pode ser realizada por meio de dois protocolos de comunicação, SPI ou SD. O SPI é um protocolo serial *bit a bit*, como visto no Capítulo 16. Já a SD trata-se de uma comunicação *nibble a nibble*, ou seja, os dados são transferidos de quatro em quatro bits, resultando um fluxo de dados superior à comunicação SPI. No entanto, a comunicação SD não está presente na maioria dos microcontroladores de baixo custo, por este motivo este capítulo está fundamentado na comunicação SPI.



O código presente neste capítulo foi testado em memórias com capacidade de até 1GB da SanDisk.

17.1 Organização da Memória

A memória do SD Card está organizada em três unidades conhecidas como *Block*, *Sector* e *WP Group*.

O *Block* (Bloco) é uma unidade relacionada aos comandos de escrita e leitura de blocos orientados. Seu tamanho corresponde à quantidade de *bytes* que serão transferidos caso o SD Card receba os comandos de leitura e escrita. O tamanho do bloco pode ser fixo ou programável e as informações sobre ele estão no registrador CSD (Dado Específico do Cartão).

O *Sector* (Setor) é uma unidade relacionada ao comando de apagar e seu tamanho corresponde à quantidade de blocos apagados. O tamanho do bloco é fixo e as informações sobre ele podem ser obtidas pelo registrador CSD (Dado Específico do Cartão).

O *WP Group* é uma unidade relacionada à proteção de dados contra escrita e seu tamanho corresponde ao número de grupos protegidos, sendo indicado por um *bit*. O tamanho do grupo é fixo e as informações sobre ele podem ser obtidas através do registrador CSD (Dado Específico do Cartão).

17.2 Registradores do SD Card

Todos os cartões SD possuem registradores de informações concernentes ao seu funcionamento. Neste capítulo serão utilizadas diversas abreviações de registradores, cujos significados podem ser vistos nesta seção.

Tabela 17.1: Registradores do SD Card.

Nome	Largura	Descrição
CID	128bits	Número de identificação do cartão. É um número individual do cartão para identificação.
CSD	128bits	Dado específico do cartão. Ele contém informações sobre as condições de operação do cartão.
OCR	32bits	Registrador de condição de operação.
RCA	16bits	Endereço relativo do cartão. é o endereço local do cartão dinamicamente sugerido pelo cartão e aprovado pelo <i>host</i> durante o processo de inicialização.
SCR	64bits	Registrador das configurações especiais do SD. ele contém informações sobre as capacidades das funções especiais do SD Card.

17.2.1 Registrador de Condição de Operação (OCR)

O registrador OCR armazena a voltagem do SD Card, que possui uma estrutura de 32bits e pode ser vista na Tabela 17.2.

Tabela 17.2: Registrador OCR.

Bit OCR	Janela de voltagem V _{DD}
0-3	Reservado
4	1.6 - 1.7V
5	1.7 - 1.8V
6	1.8 - 1.9V
7	1.9 - 2.0V
8	2.0 - 2.1V
9	2.1 - 2.2V
10	2.2 - 2.3V
11	2.3 - 2.4V

Bit OCR	Janela de voltagem V _{DD}
12	2.4 - 2.5V
13	2.5 - 2.6V
14	2.6 - 2.7V
15	2.7 - 2.8V
16	2.8 - 2.9V
17	2.9 - 3.0V
18	3.0 - 3.1V
19	3.1 - 3.2V
20	3.2 - 3.3V
21	3.3 - 3.4V
22	3.4 - 3.5V
23	3.5 - 3.6V
24 - 30	Reservado
31	Busy flag

Legenda: A área pintada indica a região de operação



Quando o bit busy for '1', significa que o procedimento de alimentação está finalizado.

17.2.2 Registrador de Identificação do Cartão (CID)

Todo cartão SD possui um número de identificação criado durante a manufatura, que não pode ser modificado. Esse registrador possui uma estrutura de 16bytes e pode ser vista na Tabela 17.3.

Tabela 17.3: Registrador CID.

Nome	Tipo	Segmento	Comentário	Exemplo
Identificação do fabricante (MID)*	Binário	[127:210]	A identificação dos fabricantes IDs é controlada e assinada pela associação SD Card.	0x03
OEM/Application ID (OID)	ASCII	[119:104]	Identifica o OEM do cartão e/ou o conteúdo do cartão.	SD ASCII Code 0x53
Nome do produto (PNM)	ASCII	[103:64]	Cinco caracteres ASCII.	SD512
Revisão do produto (PRV)**	BCD	[63:56]	Dois dígitos decimais codificados em binário. 0101 0010b = 5.2	
-	-	[23:20]	Reservado.	-
Data da manufatura (MDT)	BCD	[19:8]	Data da manufatura - aam (offset de 2000). 05 - 2005 3 - Março	0x053
CRC	Binário	[7:1]	Calculado.	xxxxxx
-	-	[0:0]	Sempre '1'.	1

Legenda: *: O valor 0x03 inclui os seguintes fabricantes: Toshiba, SanDisk e MEI.

17.2.3 Registrador de Dado Específico do Cartão (CSD)

O registrador CSD contém informações sobre as configurações requeridas para o acesso ao cartão. Esse registrador possui uma estrutura de 16bytes e pode ser vista na Tabela 17.4.

Tabela 17.4: Registrador CSD.

Nome	Tipo	Segmento	Descrição
Estrutura CSD	R	[127:126]	00b - Versão 1.0 01b a 11b - Reservado.
Reservado	R	[125:120]	00 0000b
Tempo de leitura de dado	R	[119:112]	[0:2] - Unidade de tempo começando em 1ns. Cada unidade multiplica por 10. [6:3] - Valor do tempo. 0 = Reservado, 1 = 1.0, 2 = 1.2, 3 = 1.3, 4 = 1.5, 5 = 2.0, 6 = 2.5, 7 = 3.0, 8 = 3.5, 9 = 4.0, a = 4.5, b = 5.0, c = 5.5, d = 6.0, e = 7.0, f = 8.0 [7:7] - Reservado
Tempo de leitura de dado em ciclos de clock.	R	[111:104]	Clock de referência = 25.5K. Valor multiplicado por 100.
Taxa máxima de transferência de dados	R	[103:96]	[0:2] - Unidade da taxa de transferência. 0 = 100kbit/s, 1 = 1Mbit/s, 2 = 10Mbit/s, 3 = 100Mbit/s, 4 a 7 = reservado. [6:3] - Valor do tempo. 0 = Reservado, 1 = 1.0, 2 = 1.2, 3 = 1.3, 4 = 1.5, 5 = 2.0, 6 = 2.5, 7 = 3.0, 8 = 3.5, 9 = 4.0, a = 4.5, b = 5.0, c = 5.5, d = 6.0, e = 7.0, f = 8.0 [7:7] - Reservado
Classes de comando do cartão	R	[95:84]	xxxx xxxx xxxx b
Comprimento máximo do bloco de dados para leitura	R	[83:80]	0 a 8 = reservado, 9 = 512bytes, 10 = 1024, 11 = 2048, 12 a 15 = reservado.
Permissão de leitura de bloco parcial	R	[79:79]	0 - Somente o tamanho do bloco definido por [83:80] pode ser usado para transferir dados a blocos orientados. 1 - Blocos menores também podem ser usados.
Permissão de escrita além do bloco físico	R	[78:78]	0 - Não é permitido escrever além do bloco físico 1 - É permitido escrever além do bloco físico.
Permissão de leitura além do bloco físico	R	[77:77]	0 - Não é permitido escrever além do bloco físico. 1 - É permitido escrever além do bloco físico.
Registrador do estágio de controle (DSR)	R	[76:76]	0 - DSR não implementado. 1 - DSR implementado.
Reservado	R	[75:74]	00b
Tamanho do dispositivo [Mbyte]	R	[73:62]	xxxx xxxx xxxx b
Máxima corrente para leitura na tensão de alimentação mínima	R	[61:59]	0 = 0.5mA, 1 = 1mA, 2 = 5mA, 3 = 10mA, 4 = 25mA, 5 = 35mA, 6 = 60mA, 7 = 100mA.
Máxima corrente para leitura na tensão de alimentação máxima	R	[58:56]	0 = 1mA, 1 = 5mA, 2 = 10mA, 3 = 25mA, 4 = 35mA, 5 = 45mA, 6 = 80mA, 7 = 200mA.

Nome	Tipo	Segmento	Descrição
Máxima corrente para escrita na tensão de alimentação mínima	R	[55:53]	0 = 0.5mA, 1 = 1mA, 2 = 5mA, 3 = 10mA, 4 = 25mA, 5 = 35mA, 6 = 60mA, 7 = 100mA.
Máxima corrente para escrita na tensão de alimentação máxima	R	[52:50]	0 = 1mA, 1 = 5mA, 2 = 10mA, 3 = 25mA, 4 = 35mA, 5 = 45mA, 6 = 80mA, 7 = 200mA.
Multiplicador do tamanho do dispositivo	R	[49:47]	0 = x4, 1 = x8, 2 = x16, 3 = x32, 4 = x64, 5 = x128, 6 = x256, 7 = x512
Permissão para apagar um único bloco	R	[46:46]	0 - O host pode apagar um setor definido em [45:39]. 1 - O host pode apagar um setor definido em [45:39] ou bloco definido em [25:22].
Tamanho do setor para apagar	R	[45:39]	0 = 1 bloco, 1 = 2 blocos, 2 = 3 blocos 127 = 128 blocos.
Tamanho do grupo protegido contra escrita	R	[38:32]	0 = 1 grupo, 1 = 2 grupos, 2 = 3 grupos 127 = 128 grupos blocos.
Habilita grupo protegido contra escrita	R	[31:31]	0 - Não é possível proteger nenhum grupo. 1 - É possível proteger grupo.
Reservado	R	[30:29]	00b
Fator de velocidade de escrita	R	[28:26]	0 = x1, 1 = x2, 2 = x4, 3 = x8, 4 = x16, 5 = x32, 6 a 7 = reservado
Máximo comprimento do bloco de dados para escrita	R	[25:22]	0 a 8 = reservado, 9 = 512bytes, 10 = 1024, 11 = 2048, 12 a 15 = reservado.
Permissão para escrita em blocos parciais	R	[21:21]	0 - Não. 1 - Sim.
Reservado	R	[20:16]	0 0000b
Grupo de arquivo formatado	R/W	[15:15]	0 - Sim. 1 - Não.
Sinal de cópia	R/W	[14:14]	0 - Original. 1 - Não original.
Proteção permanente contra escrita	R/W	[13:13]	0 - Não protegido. 1 - Protegido.
Proteção temporária da escrita	R/W	[12:12]	0 - Não protegido. 1 - Protegido.
Formato do arquivo	R/W	[11:10]	[21:21] = 0b e [11:10] = 00b → HD [21:21] = 0b e [11:10] = 01b → DOS FAT [21:21] = 0b e [11:10] = 10b → Formato do arquivo universal. [21:21] = 0b e [11:10] = 11b → Outros ou desconhecido.
Reservado	R/W	[9:8]	00b
CRC	R/W	[7:1]	xxxx xxxb
Não usado	-	[0:0]	1b

A capacidade de memória do cartão pode ser obtida pelo tamanho do dispositivo [73:62], multiplicador do tamanho do dispositivo [49:47] e o comprimento máximo do bloco de dados para leitura [83:80].

Sendo:

- **T_D**: tamanho do dispositivo.
- **M_T_D**: multiplicador do tamanho do dispositivo. ($M_T_D < 8$)
- **C_M_B**: comprimento máximo do bloco de dados para leitura. ($C_M_B < 12$)

Temos que:

$$\text{Capacidade_de_memória} = (T_D) * (2^{M_T_D + 2}) * (2^{C_M_B})$$

17.2.4 Registrador das Configurações Especiais (SCR)

O registrador CSD contém informações sobre as configurações especiais do SD Card. Esse registrador possui uma estrutura de 8bytes e pode ser vista na Tabela 17.5.

Tabela 17.5: Registrador SCR.

Nome	Tipo	Segmento	Descrição
Estrutura SCR	R	[63:60]	0 - Versão 1.0 1 a 15 - Reservado.
Versão	R	[59:56]	0 - Versão 1.0 1 a 15 - Reservado.
Status do dado depois de apagado	R	[55:55]	0 - Apagado. 1 - Erro.
Algoritmo suportado pelo cartão	R	[54:52]	0 - Sem segurança. 1 - Protocolo de segurança 1.0. 2 - Protocolo de segurança 2.0. 3 a 7 - Reservado.
Barramento de dado suportado	R	[51:48]	Bit 0 - 1bit (DAT0). Bit 1 - Reservado. Bit 2 - 4bits (DAT0-3). Bit 3 - Reservado.
Reservado	R	[47:32]	0
Reservado	R	[31:0]	0

17.3 Pinagem da Memória SD Card

O cartão de memória SD Card possui nove pinos, dos quais quatro são destinados à comunicação SPI (SCK, MOSI, MISO e SS), três relacionados à alimentação e o resto dos pinos não há necessidade de serem utilizados.

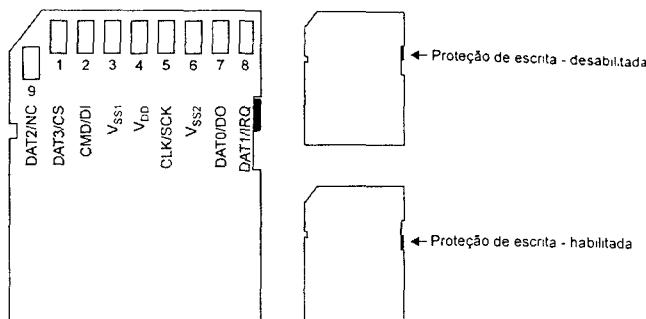


Figura 17.2: Memória SD Card padrão.

Tabela 17.6: Descrição dos pinos da memória SD Card.

Pino	Nome	Função (Modo SD)	Função (Modo SPI)
1	DAT3/CS	Linha de dado 3	Chip Select/Slave Select (\bar{SS})
2	CMD/DI	Linha de comando	Master Out Slave In (MOSI)
3	VSS1	Terra	Terra
4	VDD	Alimentação (2.7V a 3.6V)	Alimentação (2.7V a 3.6V)
5	CLK	Clock	Clock (SCK)
6	VSS2	Terra	Terra
7	DAT0/DO	Linha de dado 0	Master In Slave Out (MISO)
8	DAT1/IRQ	Linha de dado 1	Não utilizado ou IRQ
9	DAT2/NC	Linha de dado 2	Não utilizado

17.4 Modos de Instalação

As memórias SD Card funcionam com tensões que variam de 2.7V a 3.6V, e normalmente são empregados reguladores de tensão de 3.3V (por exemplo: LM1086 - 3.3V ou LM1117 - 3.3V) para alimentá-las. Entretanto, existe uma variedade de microcontroladores que operam com tensão de 5V, por isso é necessário introduzir um pequeno circuito divisor de tensão de maneira que a memória não se danifique devido à incompatibilidade de tensão.

A seguir são ilustradas duas formas de efetuar a instalação física entre um microcontrolador e um cartão SD.

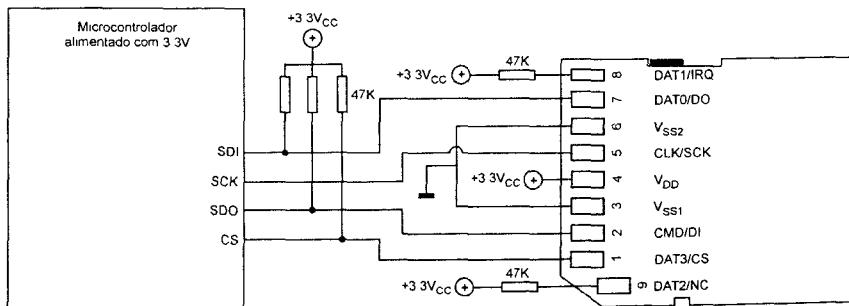


Figura 17.3: Instalação entre um microcontrolador alimentado com 3.3V e um SD Card (recomendado).

A Figura 17.3 ilustra como deve ser feita a comunicação entre os dispositivos com a mesma tensão de alimentação (3.3V). Neste caso só há necessidade de colocar resistores *pull-up* de 47K ou estar entre 10K e 100K.

Outra forma de realizar a instalação entre o microcontrolador de 5V e o cartão é mostrado na Figura 17.4. Como ambos não trabalham com a mesma tensão de alimentação, a associação dos resistores de 1K8 e 3K3 resulta um divisor de tensão, o qual reduz a tensão de 5V proveniente do microcontrolador para aproximadamente 3.24V, evitando que o cartão seja danificado pelo excesso de tensão proveniente do microcontrolador, caso não existisse esse divisor inserido no circuito.

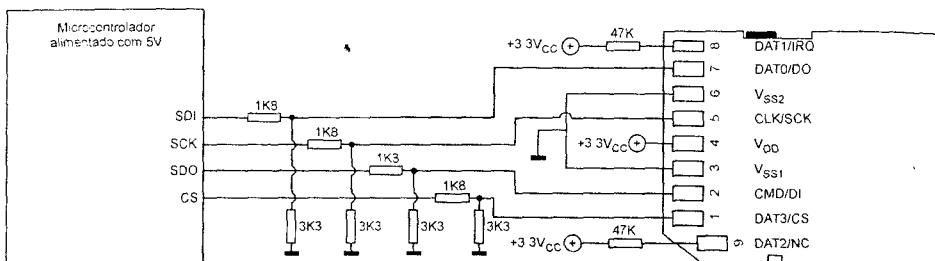


Figura 17.4: Instalação entre um microcontrolador alimentado com 5V e um SD Card.

O circuito divisor de tensão não é recomendado em projetos profissionais, pois ele distorce os sinais em frequências elevadas.

17.5 Comandos Suportados pelo SD Card

Esta seção apresenta os principais comandos suportados pelo SD Card. Os comandos podem ser básicos (prefixo CMD) ou para operações específicas (prefixo ACMD). Seja qual for o tipo de comando enviado pelo microcontrolador para o cartão, sempre haverá um tipo de resposta associada a ele.

17.5.1 Comandos Básicos

As principais funções de controle do SD Card são executadas pelos comandos listados na Tabela 17.7.

Tabela 17.7: Comandos básicos.

Índice CMD	Argumento	Resposta	Descrição
CMD0	[31:0] Irrelevante*	R1	Reinicia todos os cartões e os coloca no modo de espera.
CMD9	[31:0] Irrelevante*	R1	Solicita ao cartão selecionado enviar o seu <i>Card Specific Data</i> (CSD).
CMD10	[31:0] Irrelevante*	R1	Solicita ao cartão selecionado enviar o seu <i>Card Identification Data</i> (CID).
CMD12	[31:0] Irrelevante*	R1b	Força a parada da operação de leitura de múltiplos blocos.
CMD13	[31:0] Irrelevante*	R2	O cartão endereçado envia seu registrador de <i>status</i> .
CMD16	[31:0] Comprimento do bloco.	R1	Define o comprimento do bloco em bytes.
CMD17	[31:0] Endereço do dado	R1	Lê um bloco de dados cujo comprimento foi definido pelo comando CMD16.
CMD18	[31:0] Endereço do dado	R1	Lê blocos de dados consecutivamente até que seja interrompido por um comando CMD12 ou um novo comando de leitura.

Índice CMD	Argumento	Resposta	Descrição
CMD24	[31:0] Endereço do dado	R1	Escrive um bloco de dados cujo comprimento foi definido pelo comando CMD16.
CMD25	[31:0] Endereço do dado	R1	Escrive blocos de dados consecutivamente até que seja interrompido por um sinal <i>stop transmission</i> .
CMD27	[31:0] Irrelevante*	R1	Programa os bits programáveis do CSD (Card Specific Data)
CMD28	[31:0] Endereço do dado	R1b	Seleciona o bit de proteção contra escrita do grupo endereçado. As propriedades de proteção contra escrita são codificadas no dado específico do cartão.
CMD29	[31:0] Endereço do dado	R1b	Limpia o bit de proteção contra escrita do grupo endereçado.
CMD30	[31:0] Endereço do dado protegido contra escrita	R1	Solicita ao cartão para enviar o status dos bits de proteção contra escrita.
CMD32	[31:0] Endereço do dado	R1	Seleciona o endereço do primeiro bloco de escrita a ser apagado.
CMD33	[31:0] Endereço do dado	R1	Seleciona o endereço do último bloco de escrita a ser apagado.
CMD38	[31:0] Irrelevante*	R1b	Apaga todos os setores ou grupos anteriormente selecionados.
CMD55	[31:0] Complemento	R1	Informa ao cartão que o próximo comando trata-se de um comando para aplicações específicas (ACMD).
CMD56	[31:1] Complemento [0] RD/WR	R1	Usado para transferir um bloco de dados para o cartão ou receber um bloco de dados do cartão para comandos de propósito geral ou específico.
CMD58	[31:0] Irrelevante*	R3	Lê o registrador OCR (Registrador de Condição de Operação) do cartão.
CMD59	[31:1] Irrelevante* [0] Opção CRC	R1	Liga/desliga o CRC (Verificação Ciclica Redundante). '0' - desliga '1' - liga

Legenda: * - O lugar do bit deve ser preenchido. No entanto, o valor é irrelevante

17.5.2 Comandos Específicos

As funções específicas do SD Card são executadas pelos comandos listados na Tabela 17.8.

Tabela 17.8: Comandos especiais.

Índice ACMD	Argumento	Resposta	Descrição
ACMD13	[31:0] Complemento	R2	Envia o status do cartão.
ACMD22	[31:0] Complemento	R1	Envia o número de blocos escritos com sucesso. Resposta com 32bits+CRC.
ACMD23	[31:23] Complemento [22:0] número de blocos	R1	Seleciona o número de blocos escritos para ser pré-apagado antes de escrever. O valor '1' é padrão e corresponde a um bloco.
ACMD41	[31:0] Irrelevante*	R1	Ativa o processo de inicialização do cartão.
ACMD42	[31:1] Complemento [0] set_cd	R1	Conecta ou desconecta o resistor <i>pull-up</i> no pino 1 (CS/DAT3). Esta configuração é feita através do parâmetro <i>set_cd</i> , cujo valor pode ser: '1' - Conecta '0' - Desconecta
ACMD51	[31:0] Complemento	R1	Lê o SCR (<i>SD Configuration Register</i>) do SD Card.

Legenda: * - O lugar do bit deve ser preenchido. No entanto, o valor é irrelevante.

17.6 Formato Padrão da Comunicação

Todo comando é constituído de 6bytes. O primeiro byte contém o 1bit de início (*Start*) + 1bit indicador do transmissor + 6bits que indicam se o comando é básico ou específico, do segundo ao quinto byte correspondem ao argumento do comando e o último byte é constituído de 7bits de Verificação Cíclica Redundante (CRC) + 1bit que indica o fim da transmissão (*End*).

O início e o fim do comando são indicados pelos bits de *Start* e *End*, cujos valores são constantes '0' e '1' respectivamente. A origem da transmissão é informada através do bit indicador do transmissor. Se esse bit for '1', indica que o comando foi enviado pelo *host* (neste caso é o microcontrolador), e se for '0', indica que é uma resposta do cartão.

Desta forma, o formato do comando pode ser representado de acordo com a Figura 17.5.

Byte 1				Byte 2 - 5				Byte 6		
7	6	5	0	31		0	7	1	0	
Start	Transmissor	Comando		Argumento do comando				CRC	End	

Figura 17.5: Formato do comando.

O comando enviado pelo microcontrolador tem o formato mostrado na Figura 17.6.

Byte 1				Byte 2 - 5				Byte 6		
7	6	5	0	31		0	7	1	0	
0	1	Comando		Argumento do comando				CRC	1	

Figura 17.6: Formato do comando enviado pelo *host*.

Os formatos das respostas R1 e R3 do cartão são similares ao comando enviado pelo host, como pode ser observado pela Figura 17.6. Já a resposta R2 é mais longa do que as demais, e os conteúdos transmitidos são o CID (*Card Identification Data*) e o CSD (*Card-Specific Data*).

Quando o cartão está configurado para funcionar no modo SPI, a Checagem Cíclica Redundante (CRC) está desabilitada como padrão, por este motivo não há necessidade de fazer a verificação, porém ele deve ser enviado/recebido. Somente o comando CMD0 deve ter o valor de CRC enviado adequadamente, pois trata-se de um comando estático e sempre gera o mesmo CRC, que em conjunto com o bit *End* resulta em um byte equivalente a 0x95. Logo, o byte 6 do comando CMD0 é igual a 0x95.

Os comandos CMD e ACMD ocupam os seis bits menos significativos do byte 1. Como os dois bits mais significativos são constantes, o valor do byte 1 é definido por meio do cálculo seguinte:

- **Comando CMD0:** Byte 1 = $0x40 \mid 0x00 = 0x40$.
- **Comando CMD17:** Byte 1 = $0x40 \mid 0x11 = 0x51$.
- **Comando ACMD41:** Byte 1 = $0x40 \mid 0x29 = 0x69$.

17.7 Respostas dos Comandos

Todo comando enviado do dispositivo *master* (microcontrolador) para o *slave* (SD Card) possui uma resposta peculiar a ele, como se observa nas Tabelas 17.7 e 17.8. Elas podem ser identificadas como R1, R1b, R2 e R3, e desempenham um papel fundamental na comunicação entre os dispositivos.

17.7.1 Resposta R1

A resposta R1 é enviada pelo SD Card após a maioria dos comandos. Ela possui um tamanho de 8 bits, cujos erros são sinalizados por '1'.

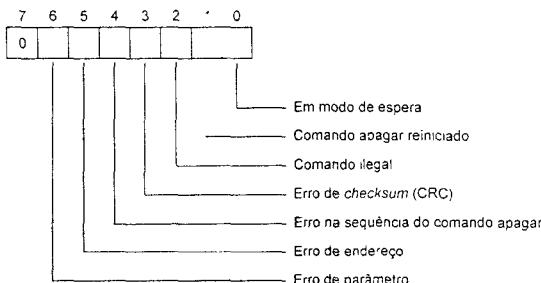


Figura 17.7. Formato da resposta R1.

17.7.2 Resposta R1b

O formato dessa resposta é idêntico ao R1, sendo a única diferença a existência de um *busy flag*, em que '0' indica que o cartão está ocupado e '1' indica que o cartão está pronto para o próximo comando.

17.7.3 Resposta R2

Essa resposta é constituída de 2bytes. Ela é enviada como resposta para o comando ACMD13, e seu formato pode ser visto na figura a seguir.

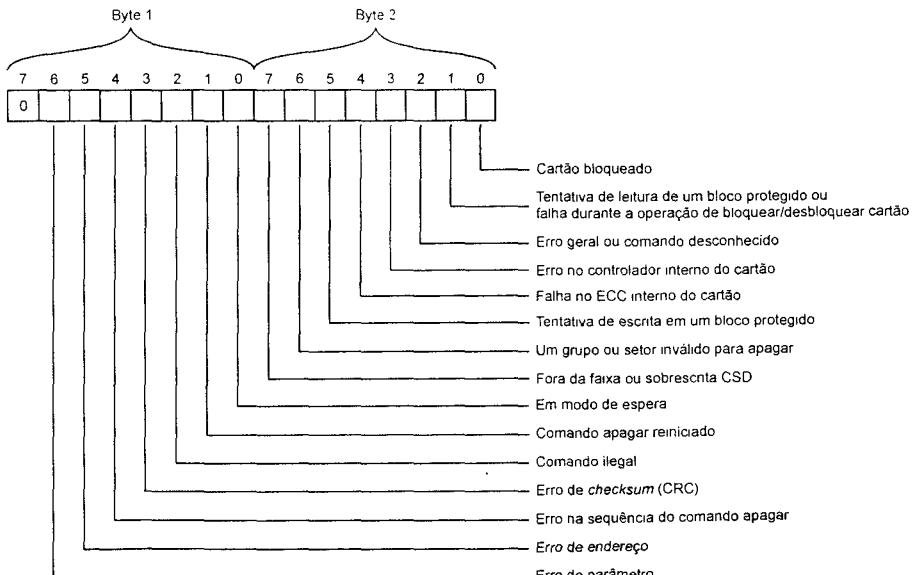


Figura 17.8. Formato da resposta R2.

17.7.4 Resposta R3

Essa resposta possui um comprimento de 5bytes e é enviada pelo cartão quando o comando CMD58 é recebido. A estrutura do primeiro byte é idêntica ao tipo de resposta R1. Os outros 4bytes representam o registro OCR (*Operation Condition Register*).

39	32	31	0
Formato da resposta R1		OCR	

Figura 17.9: Formato da resposta R3.

17.8 Sinais Relacionados aos Dados

17.8.1 Sinal de Início e Parada de Transmissão

Os sinais de dados estão presentes em todas as operações de escrita e leitura de blocos. Eles são constituídos de 4 a 515bytes e possuem dois formatos, que serão apresentados a seguir.

A leitura de único bloco (CMD17), escrita de único bloco (CMD24) e leitura de múltiplos blocos (CMD18), antes de iniciarem a transmissão dos dados no bloco selecionado, primeiramente devem enviar um sinal de início, conhecido como *Start block* (valor 0xFE), seguido dos dados que compõem a informação desejada. Os dois últimos bytes são utilizados para verificação de erro (CRC). Seu formato pode ser visualizado na Figura 17.10.

Byte 1	Byte 2 ...	Límite - Byte 513	Dois últimos bytes enviados
0xFE	Dados que constituem a informação		CHECKSUM (CRC)

Start block

Figura 17.10: Formato do sinal para escrita/leitura de único bloco e leitura de múltiplos blocos.

O comando de leitura de múltiplos dados pode ser interrompido enviando o comando de parada de transmissão (CMD12).

O segundo formato possível está relacionado à operação de escrita de múltiplos blocos. Ele é semelhante ao primeiro formato, no entanto percebe-se que o sinal de *Start block* (valor 0xFC) não possui o mesmo valor e a interrupção do comando ocorre através de um sinal de parada (*Stop transmission*) cujo valor é 0xFD.

Byte 1	Byte 2 ...	Byte n
0xFC	Dados que constituem a informação	0xFD

Start block

Stop transmission

Figura 17.11: Formato do sinal para escrita de múltiplos blocos.

17.8.2 Sinal de Status da Escrita de Dado

Sempre que o comando de escrita de dados for utilizado, seja o comando de escrita de um único bloco (CMD24) ou múltiplos blocos (CMD25), o cartão retorna um sinal informando se a operação foi bem-sucedida ou não. Esse sinal possui um tamanho de 8bits, cujo formato está ilustrado na Figura 17.12.

7	6	5	4	3	1	0
x	x	x	0	Status		1

Figura 17.12: Formato do status do dado escrito.

O parâmetro *Status* ocupa 3 bits do sinal de status de escrita de dados e pode assumir três valores distintos, que fornecem informações relacionadas à operação de escrita. O significado dos valores pode ser interpretado de acordo com a lista seguinte:

- **010:** dado aceito.
- **101:** dado rejeitado devido ao erro de CRC.
- **110:** dado rejeitado devido ao erro de escrita.

Se durante a operação de escrita de múltiplos blocos ocorrer um erro, a operação pode ser interrompida através do comando de parada de transmissão (CMD12) e os blocos escritos com sucesso podem ser verificados pelo comando ACMD22. Caso seja gerado um erro de escrita ('110'), é possível conhecer a sua origem pelo comando de status CMD13.

17.8.3 Sinal de Erro de Dado

Toda vez que uma falha na operação ocorrer e o dado solicitado não puder ser providenciado pelo SD Card, ele retorna um sinal indicando o erro. Esse sinal de erro possui um tamanho equivalente a 8bits, dos quais os 4 bits mais significativos (*nibble* MSB) são constantes e os 4 bits menos significativos (*nibble* LSB) informam a origem do problema. O seu formato pode ser visualizado na Figura 17.13.

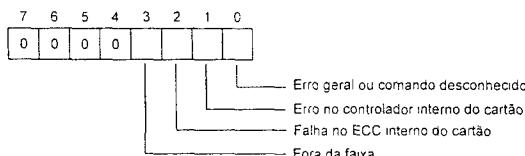


Figura 17.13: Formato do sinal de erro de dado.

17.9 Operações para Ler e Escrever Dados

Cartões SD Card suportam operações de leitura e escrita em um único bloco ou em múltiplos blocos. No entanto, há um limite de tempo para as operações, por exemplo: a operação de leitura deve ser inferior a 100ms, já as operações para escrever dados devem ocorrer em um período inferior a 250ms. Se a operação ultrapassar o limite de tempo estabelecido, o cartão retorna uma mensagem de erro.



Após a última transação no barramento SPI, ou seja, após receber o *End bit*, o host (microcontrolador) deve enviar obrigatoriamente oito *clocks* para finalizar a operação.

17.9.1 Operação de Leitura de Um Bloco

A operação de leitura de um único bloco de dados é iniciada a partir do envio do comando CMD17, com o endereço do bloco do qual se deseja efetuar a leitura, mais o *checksum* (CRC). Após receber o comando, o cartão retorna uma resposta R1 cujo valor deve ser igual a 0x00, indicando a não ocorrência de erros no comando. Ao receber a resposta R1= 0x00, o microcontrolador aguarda o sinal de início do bloco (*start block* = 0xFE), então efetua a leitura de todos os dados contidos no bloco selecionado, acrescido do *checksum* (CRC).

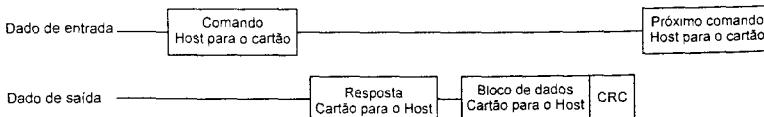


Figura 17.14: Operação de leitura de um bloco.

17.9.2 Operação de Leitura de Múltiplos Blocos

A operação de leitura de múltiplos blocos de dados é iniciada a partir do envio do comando CMD18, com o endereço do bloco do qual se deseja efetuar a leitura, mais o *checksum* (CRC). Após receber o comando, o cartão retorna uma resposta R1 cujo valor deve ser igual a 0x00, indicando a não ocorrência de erros no comando. Ao receber a resposta R1=0x00, o microcontrolador aguarda o sinal de inicio do bloco (*start block* = 0xFE), então efetua a leitura do bloco selecionado. Caso o dispositivo queira ler o próximo bloco, ele aguarda novamente o sinal de inicio do bloco, lê os dados e assim por diante. Quando quiser sair do comando, o dispositivo envia o comando CMD12 e finaliza a operação.

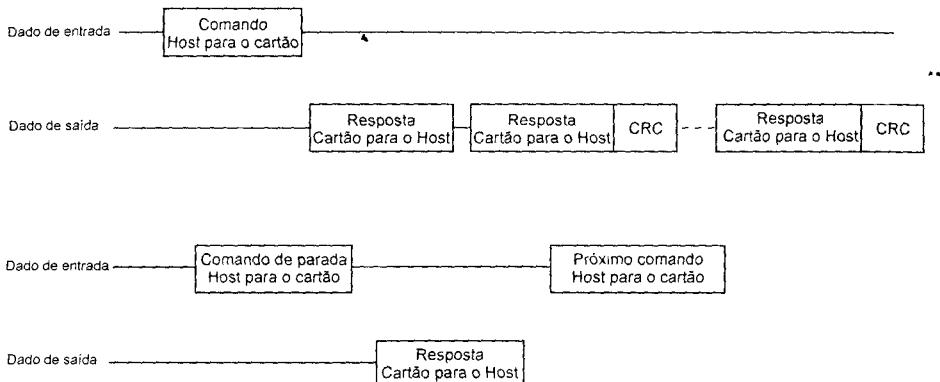


Figura 17.15: Operação de leitura de múltiplos blocos.

17.9.3 Operação de Escrita em Um Bloco

A operação de escrita de um único bloco de dados é iniciada a partir do envio do comando CMD24, com o endereço do bloco do qual se deseja efetuar a escrita, mais o *checksum* (CRC). Após receber o comando, o cartão retorna uma resposta R1 cujo valor deve ser igual a 0x00, indicando a não ocorrência de erros no comando. Em seguida, o dispositivo envia o sinal de inicio do bloco (*start block* = 0xFE), então efetua a escrita no bloco selecionado. Ao receber todos os dados, a memória SD Card retorna o seu status cujo valor deve ser (010 - dado aceito), então finaliza a operação.

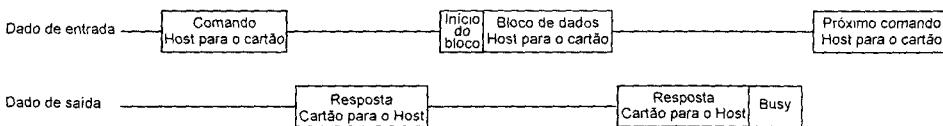


Figura 17.16: Operação de escrita de um bloco.

17.9.4 Operação de Escrita em Múltiplos Blocos

A operação de escrita em múltiplos blocos de dados é iniciada a partir do envio do comando CMD25, com o endereço do bloco do qual se deseja efetuar a escrita, mais o *checksum* (CRC). Após receber o comando, o cartão retorna uma resposta R1 cujo valor deve ser igual a 0x00, indicando a não ocorrência de erros no comando. Em seguida, o dispositivo envia o sinal de inicio do bloco (*start block* = 0xFC), então efetua a escrita no bloco selecionado. Ao receber todos os dados, a memória SD Card retorna o seu status cujo valor deve ser

(010 - dado aceito). Caso o dispositivo queira escrever no próximo bloco, ele envia novamente o sinal de inicio do bloco, escreve os dados, lê o status e assim por diante. Quando quiser interromper o comando, o dispositivo envia o sinal de parada de transmissão (*stop tran*), então finaliza a operação.

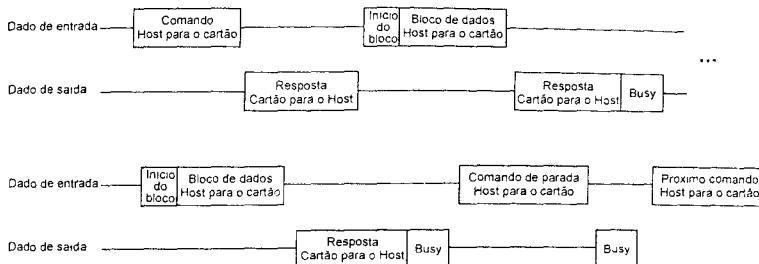


Figura 17.17: Operação de escrita de múltiplos blocos.

17.10 Inicialização do SD Card

A sequência de inicialização do SD Card começa pelo procedimento de carregamento da tensão de alimentação. Ele é feito a partir do envio de uma sequência de nível lógico '1' pelo canal CMD equivalente a 80 *clocks*. Esse atraso tem a função de estabilizar a tensão e evitar problemas de comunicação. Em seguida é enviado o comando para reiniciar o SD Card (CMD0). Após esse comando o cartão deve ser informado que uma operação específica será enviada, então o comando CMD55 prepara o cartão para receber o comando ACMD41, que fará a ativação do processo de inicialização. Por fim é necessário verificar se a tensão de alimentação está dentro da faixa de operação do SD Card. Caso esteja dentro dessa faixa, o *clock* é selecionado para a máxima frequência de operação, que não pode exceder a máxima suportada pelo cartão.

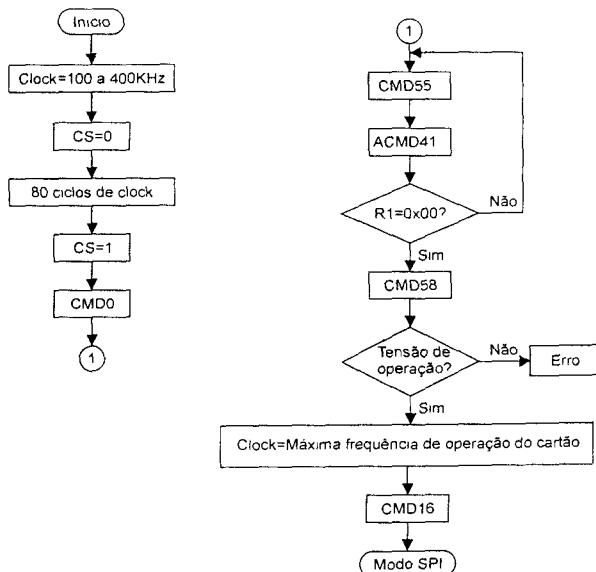


Figura 17.18: Fluxograma para inicialização do SD Card.

17.11 Biblioteca do Cartão SD Card

A biblioteca de manipulação do SD Card utilizado neste livro está localizada no arquivo disponibilizado pelo autor (**biblioteca_sd_card.h**) no site da Editora Érica (www.editoraerica.com.br). Para que esta biblioteca funcione adequadamente é necessário definir dois parâmetros, o tamanho do bloco e o pino de controle, dentro do programa.

/ O tamanho do bloco pode ser definido como 32, 64, 128, 256bytes ... Porém, não são todos os cartões que suportam estes comprimentos, o que pode ser verificado no Datasheet do cartão ou experimentalmente.*/*

#define TAMANHO_BLOCO 512 //Define o tamanho do bloco em bytes.

//Este pino pode ser qualquer pino I/O digital, e deve ser configurado como saída.

#define ENABLE_SD PORTDbits.RD4 //Define o pino destinado a habilitar o SD Card.

Vejamos na sequência, a lista de comandos disponibilizados pelo arquivo.

Tabela 17.9: Descrição das funções contínuas na **biblioteca_sd_card.h**.

Função	Descrição
inicia_sd ()	Inicia o SD Card.
escreve_bloco (bloco, dados)	Insere dados no bloco de dados especificado. Sendo: <i>bloco</i> : Número do bloco. <i>dados</i> : Ponteiro para a localização dos dados.
escreve_seq_bloco (bloco, quanti_blocos)	Insere dados em uma sequência de blocos Sendo: <i>bloco</i> : Número do bloco inicial. <i>quanti_blocos</i> : Quantidade de blocos que devem ser escritos. Observação: Deve ser alterado pelo usuário.
le_bloco (bloco, dados)	Efetua a leitura dos dados contidos no bloco especificado. Sendo: <i>bloco</i> : Número do bloco. <i>dados</i> : Ponteiro para a localização dos dados.
le_seq_bloco (bloco, quanti_blocos)	Efetua a leitura dos dados contidos em uma sequência de blocos. Sendo: <i>bloco</i> : Número do bloco inicial. <i>quanti_blocos</i> : Quantidade de blocos que devem ser lidos. Observação: Deve ser alterado pelo usuário.

Observação: Todas as funções retornam 0x01 para indicar que a operação foi realizada com sucesso. Caso contrário, se retornar 0x00 indica que ocorreu um erro na operação.

17.12 Projeto

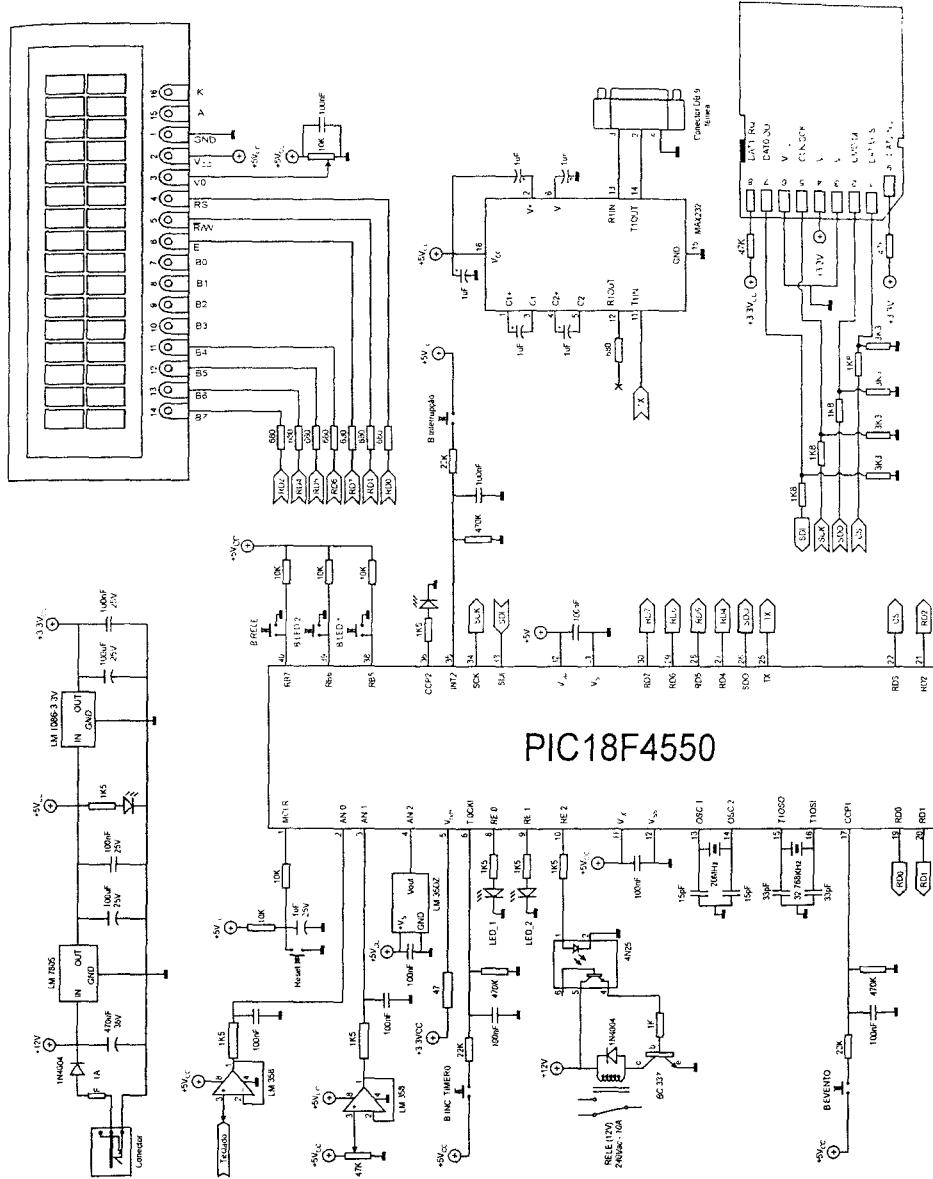


Figura 17.19: Circuito para comunicação com SD Card.

Código

```

***** Projeto Capítulo 17 - SD Card *****
**
** Este programa faz a leitura do sensor de temperatura LM35D (AN2) com um tempo de amostragem equivalente a 200ms.
** Após obter 512 amostras, os dados são gravados na SD Card, e enviados para a RS-232.
** O programa pode ser interrompido a qualquer momento, bastando pressionar o botão ligado ao pino RB7.
**
** Autor: Alberto Noboru Miyadaira
*****



#include <18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <delays.h> //Adiciona a biblioteca de funções de delay.
#include <spi.h> //Adiciona a biblioteca de funções da comunicação SPI, implementada em hardware.
#include <adc.h> //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <stdio.h> //Adiciona a biblioteca padrão de entrada e saída.
#include <stdlib.h> //Adiciona a biblioteca de funções miscelâneas.
#include <string.h> //Adiciona a biblioteca de manipulação de string.
#include <math.h> //Adiciona a biblioteca de funções matemática.
#include "C:\pic18\rotina_Lcd_2x16.h" //Biblioteca contendo as funções do LCD.
#include "C:\pic18\config_PIC18F4550.h" //Adiciona a configuração do microcontrolador PIC18F4550. (Capítulo 6)

//Estes dois protótipos devem ser inseridos antes da função SD Card, pois serão usados por ela.
void imprime_char_RS232 (char); //Protótipo.
void imprime_string_RS232 (const rom char *); //Protótipo.

/*
* O tamanho do bloco pode ser definido como 32, 64, 128, 256bytes ... Porém, não são todos os cartões que suportam
estes comprimentos, o que pode ser verificado no Datasheet do cartão ou experimentalmente.*/
#define TAMANHO_BLOCO 512 //Define o tamanho do bloco em bytes.

//Este pino pode ser qualquer pino I/O digital, e deve ser configurado como saída.
#define ENABLE_SD PORTDbits.RD4 //Define o pino destinado a habilitar o SD Card.

/*
* Se o tamanho do bloco for superior a 256 bytes, será necessário declarar duas matrizes de 256 posições cada, cuja localização na
memória de dados deve obrigatoriamente obedecer uma sequência. Por exemplo: se a primeira matriz estiver em 0x400 a segunda deve
estar em 0x500 e assim por diante.*/
#pragma udata meu_dados_1=0x100
    char dados_prov_sd1 [256];
#pragma udata meu_dados_2=0x200
    char dados_prov_sd2 [256];
#pragma udata

//Adiciona a biblioteca contendo as funções de controle do SD Card.
#include "C:\pic18\rotina_sd_card.h"

#define B_INTERROMPE PORTBbits.RB7 //Pino definido para interromper o programa.
#define LÉD_1 PORTEbits.RE0 //Indica que os dados da temperatura estão sendo inseridos no cartão SD Card.
#define LED_2 PORTEbits.RE1 //Indica que o cartão SD Card está sendo lido.
#define TX_RS232 PORTCbits.RC6 //Pino definido para transmissão dos dados, via serial.

unsigned char buffer[16];

void tempo_bit_RS232 () //Taxa = 9600bps.
{
    Delay100TCYx (5); //Gera um delay de 500 ciclos de máquina. T = 4*500/20M = 100us.
    Delay10TCYx (2); //Gera um delay de 20 ciclos de máquina. T = 4*20/20M = 4us.
}

//Adaptado para comunicação serial RS-232 com taxa de 9600bps, sendo Fosc=20MHz.
void imprime_char_RS232 (char dado_rs232)

```

```

{
    TRISCbits.TRISC6=0;//Define o pino RC6 como saída

    TX_RS232=0; //Start
    tempo_bit_RS232();

    TX_RS232=dado_rs232&0x01; //Bit 0
    tempo_bit_RS232();
    TX_RS232=(dado_rs232>>1)&0x01; //Bit 1
    tempo_bit_RS232();
    TX_RS232=(dado_rs232>>2)&0x01; //Bit 2
    tempo_bit_RS232();
    TX_RS232=(dado_rs232>>3)&0x01; //Bit 3
    tempo_bit_RS232();
    TX_RS232=(dado_rs232>>4)&0x01; //Bit 4
    tempo_bit_RS232();
    TX_RS232=(dado_rs232>>5)&0x01; //Bit 5
    tempo_bit_RS232();
    TX_RS232=(dado_rs232>>6)&0x01; //Bit 6
    tempo_bit_RS232();
    TX_RS232=(dado_rs232>>7)&0x01; //Bit 7
    tempo_bit_RS232();

    TX_RS232=1;//Stop
    tempo_bit_RS232();
    tempo_bit_RS232();
}

//Envia uma string (localizada na memória de dados) para a RS-232
void imprime_buffer_RS232(unsigned char *s_char_rs232, unsigned int tamanho_buffer)
{
    while (tamanho_buffer--)
    { imprime_char_RS232(*s_char_rs232) s_char_rs232++; }

}

//Envia uma string (localizada na memória de programa) para a RS-232.
void imprime_string_RS232(const rom char *s_char_rs232)
{
    while (*s_char_rs232)
    { imprime_char_RS232(*s_char_rs232); s_char_rs232++; }

}

//Converte o valor devolvido pelo conversor em um valor correspondente à tensão [mV]
float converte_tensao (float valor_conversor)
{ return (valor_conversor*3300)/1023; } //Neste caso, a tensão de entrada no pino Vref+ é 3.3V.

//Converte o valor de tensão em um valor correspondente à temperatura [C].
//Sensor de temperatura: LM35D - 10mV/C. Range de temperatura -55C a +150C
float converte_temperatura (float valor_tens)
{ return valor_tens/10; }

//Filtro em software.
//Obtém 256 amostras e retorna a média.
unsigned int filtro_canal()
{
    unsigned int cont_filtro;
    unsigned long valor_canal = 0;

    for( cont_filtro = 0 ; cont_filtro < 256 ; cont_filtro++ )
    {
        ConvertADC(); //Inicia a conversão.
        while (BusyADC()); //Aguarda o fim da conversão.
    }
}

```

```

    valor_canal += ReadADC( );           //Armazena o resultado da conversão.
}
return (valor_canal >> 8); // Esta operação é igual a valor_canal/256.
}

void main ()//Função principal.
{
//Variáveis locais.
unsigned char status_grava;
float valor;                         // Armazena o valor devolvido pelo conversor A/D.
float valor_temperatura;             //Armazena o valor da temperatura (LM35D)
unsigned char t_sd;                  //Registra a quantidade de tentativas de inicialização do SD-CARD.
long cont;                           //Contador de amostras
long n_bloco;                        //Armazena o número do bloco a ser gravado.

TRISA = 0b00011111; //RA0 a RA4 – entrada e RA5 a RA6 – saída.
TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 – entrada e RB3 a RB4 – saída.
TRISC = 0b00111111; //RC0 a RC5 e RC7 – entrada e RC6 – saída.
TRISD = 0b00000000; //RD0 a RD7 – saída.
TRISE = 0b00000000; //RE0 a RE2 – saída.

```

```

OpenADC (ADC_FOSC_16 //Seleção da fonte de clock para a conversão A/D. Tad = 16/20M = 0,8us.
&ADC_RIGHT_JUST          //Resultado justificado para a direita.
&ADC_4_TAD,               //Configuração do tempo de aquisição automático. (4*Tad = 3.2us)
ADC_CH2                   //Seleciona o canal 2 (AN2)
&ADC_INT_OFF              //Interrupção desabilitada.
&ADC_VREFPLUS_EXT        //Vref+ = Vref+ (pin AN3)
&ADC_VREFMINUS_VSS,       //Vref- = Vss
ADC_6ANA);                //Habilita o canal AN0 a AN5.

```

Delay10TCYx(5); // Delay de 50 ciclos de máquina.

PORTE = 0x00; //Coloca a porta D em 0V.

PORTE = 0x00; //Coloca a porta E em 0V.

lcd_inicia(0x28, 0xF, 0x06); //Iniciaiza o display LCD alfanumérico com quatro linhas de dados.

lcd_LD_cursor (0); // Desliga o cursor.

```

lcd_posicao (1,1); //Desloca o ponteiro para a L=1 e C=1.
imprime_string_lcd(" DATALOGGER "); //Imprime uma string no display.
imprime_string_RS232("Iniciando SD Card.\n"); //Envia a string para a RS-232.

```

```

for(t_sd=0;t_sd<10;t_sd++)
{
    if(inicia_sdc()==1) //Chama a função de inicialização do SD-CARD e verifica o status do cartão.
        break; //Força a saída do laço.
}

if (t_sd == 10)
{
    imprime_string_RS232 ("Erro na inicialização do SD Card.\n"); //Envia a string para a RS-232.
    lcd_posicao (2,1); //Desloca o ponteiro para a L=2 e C=1.
    imprime_string_lcd(" ERRO SD Card "); //Imprime uma string no display.
}
else
{
    status_grava = 0; //Flag para gravação.

    SetChanADC (ADC_CH2); //Carrega o canal analógico 2. (AN2) - Temperatura.

    n_bloco = 0; //Inicia a gravação no Bloco 0.
}

```

```

while (!status_grava)
{
    /* Incrementa em 8, pois a cada iteração 8 bytes são escritos no buffer (dados_prov_sd1 ou dados_prov_sd2);
    for( cont=0; cont<TAMANHO_BLOCO && !status_grava, cont+=8)
    {
        valor = filtro_canal (); //Chama a função para fazer a leitura do canal

        //Converte o valor retornado pelo conversor A/D em um valor de tensão correspondente.
        valor_temperatura = converte_tensao(valor);
        valor_temperatura = converte_temperatura (valor_temperatura); //Converte a tensão em temperatura.

        //Envia a string formatada para o buffer.
        sprintf(buffer, "T%03d.%02d.", (char)valor_temperatura.(char)((valor_temperatura-(char)valor_temperatura)*100));
        memcpys(dados_prov_sd1 + cont, buffer, 8);

        Delay10KTCYx (100); //Tempo de amostragem = 200ms. T = (4*1.000000)/Tosc = 200.000us

        if(B_INTERROMPE==0) //Se o sinal na entrada do pino RB7 for igual a 0V, interrompe o programa.
        {
            imprime_string_RS232("\nPrograma Finalizado \n"); //Imprime uma string
            status_grava = 1; //Flag para gravação.
        }
    }
    LED_1=1;
    imprime_string_RS232("\nGravando dados no BLOCO.. \n"); //Imprime uma string
    if(escreve_bloco (nBloco, dados_prov_sd1)) //Grava os valores da temperatura no SD Card.
        imprime_string_RS232("\nGravação Finalizada."); //Imprime uma string.
    else
        imprime_string_RS232("\nErro no processo de escrita."); //Imprime uma string

    LED_1=0;
    LED_2=1;

    imprime_string_RS232("\nLendo dados do BLOCO\n"), //Imprime uma string.
    if(lE_bloco (nBloco, dados_prov_sd1)) //Lê os valores da temperatura armazenados no SD Card.
    {
        imprime_string_RS232("\nLeitura finalizada."); //Imprime uma string.
        imprime_buffer_RS232(dados_prov_sd1, 512);
    }
    else
        imprime_string_RS232("\nErro no processo de leitura."); //Imprime uma string

    LED_2=0;

    lcd_posicao (2,1); //Desloca o ponteiro para a L=2 e C=1.
    sprintf(buffer, "[000000:%06lu]", nBloco); //Envia a string formatada para o buffer.
    imprime_buffer_lcd(buffer,15); //Coloca no visor, os 15 elementos do buffer.

    nBloco++; //Incrementa o bloco a cada iteração do laço while
}
lcd_posicao (2,1); //Desloca o ponteiro para a L=2 e C=1.
imprime_string_lcd("FIM DO PROGRAMA "); //Imprime uma string no display.
}

Sleep(); //Coloca o dispositivo no modo Sleep.
}

```

18

USB (Universal Serial Bus)

O USB (Barramento Serial Universal) é um dos barramentos mais utilizados para prover comunicação de alta velocidade entre periféricos e o PC (Computador Pessoal). Sendo assim, não poderia deixar de ser mencionado nesta obra.

Este capítulo faz uma breve introdução ao barramento e ao protocolo USB, abordando alguns conceitos básicos de seu funcionamento. Ele está fundamentado no microcontrolador PIC18F4550, que possui um módulo USB 2.0 interno, cujas características serão comentadas no decorrer deste capítulo.

Informações adicionais sobre a especificação USB podem ser obtidas no site www.usb.org.

18.1 Introdução

O USB foi projetado para simplificar a comunicação entre periféricos e o computador (PC). Trata-se de uma comunicação assíncrona padronizada, que opera no modo *half-duplex*, com sinal diferencial e codificação NRZI (*Non Return to Zero Invert*).

O barramento é relativamente simples, sendo composto de um terra (GND), um V_{CC} (5V) e duas linhas de dados D- e D+. Ele é capaz de prover tensão de alimentação de 5V aos dispositivos de baixo consumo, suporta altas taxas de comunicação (na versão 2.0 *High-Speed* pode chegar a 480Mbps), não há necessidade de desligar o PC para conectar/desconectar o dispositivo USB, é *plug-and-play* e suporta até 127 dispositivos.

A grande vantagem dessa interface com relação às outras é que os produtos (periféricos, *hubs*, conectores, entre outros) são gerenciados pela corporação USB-IF (*Implementers Forum*), responsável pela verificação da conformidade do produto com a especificação USB. Em outras palavras, a USB-IF é responsável pela realização de testes para verificar o nível de aceitação do produto, e caso passe nos testes, recebe um certificado USB-IF.



A corrente máxima que um *root hub* ou *hub* com alimentação própria pode fornecer é 500mA. Se o dispositivo USB estiver conectado em um *hub* alimentado pelo barramento, não deve consumir mais que 100mA.

18.2 Topologia USB

A topologia USB pode ser analisada por camadas. No topo da camada temos o *host* (hospedeiro) com um *root hub* (*hub* raiz) integrado, que pode ser um PC ou um dispositivo dotado de um USB *On-The-Go*, que ora pode se comportar como *host* ou dispositivo (*device*).

O host é o elemento responsável pela inicialização dos serviços em um barramento USB e comunica-se com um *hub* (dispositivo que dispõe de pontos de ligação adicionais para a USB) ou um ou mais dispositivos periféricos através do *root hub*. Veja a Figura 18.1.

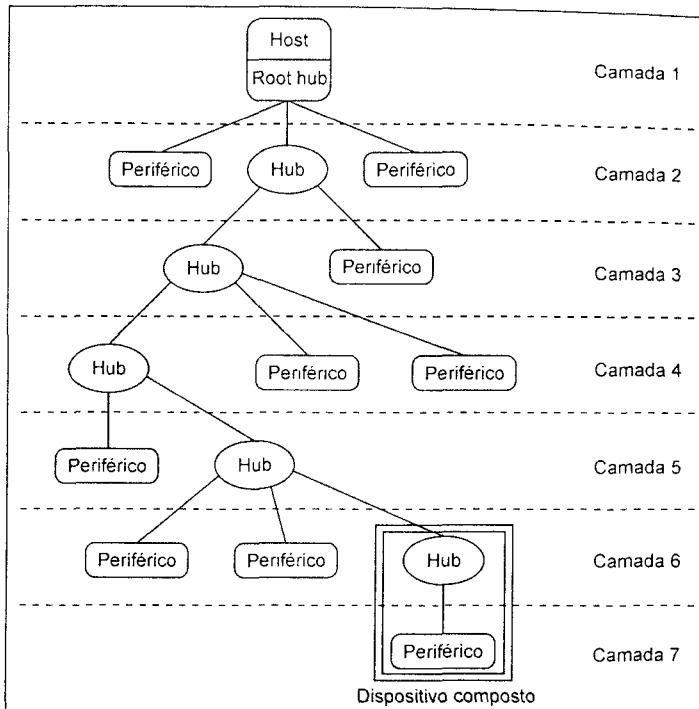


Figura 18.1: Topologia USB.

O barramento USB suporta no máximo sete camadas, e o dispositivo composto (*Compound Device*), equipado com um *hub* e periféricos, não pode ser conectado na camada sete, pois ele ocupa duas camadas.

18.3 Pinagem dos Conectores Padrão

Os *plugs* e conectores padrão da interface USB são A, B, Mini-B, Micro-AB e Micro-B. A Figura 18.2 ilustra os conectores A e B.

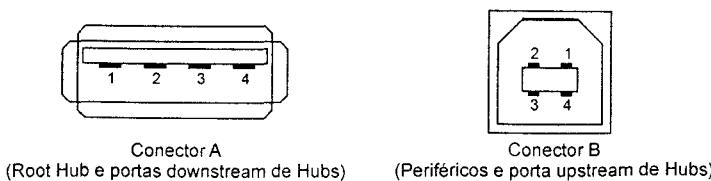


Figura 18.2: Conectores USB dos tipos A e B.

Tabela 18.1A: Nomenclatura dos pinos.

Número do pino	Nome	Cor	Descrição
1	V _{BUS}	Vermelho	Tensão do barramento ($+5V \pm 5\%$)
2	D-	Branco	Linha D-
3	D+	Verde	Linha D+
4	GND	Preto	Terra

A série A é utilizada para conectar dispositivos USB em portas *downstreams* (orientadas para baixo) de um *hub* ou *root hub*, enquanto a série B é usada nas portas *upstreams* (orientadas para cima) de um *hub* ou periféricos. Veja a Figura 18.3.

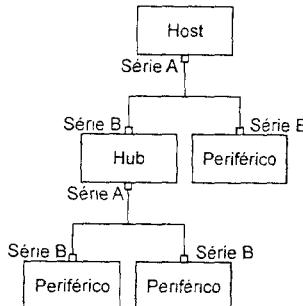


Figura 18.3: Esquema de conectores.

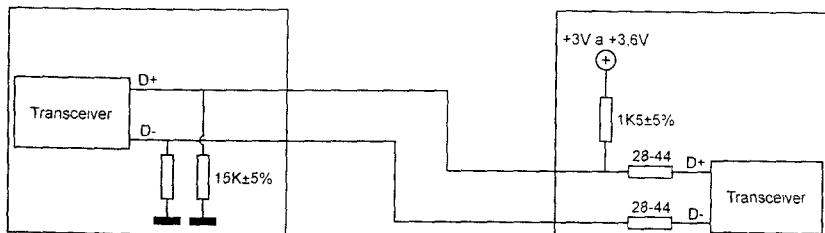
18.4 Taxas de transferência Suportadas pelo USB

O barramento USB pode trabalhar em quatro taxas de transferências distintas. São elas:

- **Low-Speed (USB 1.0)**: estima uma velocidade de 1.5Mbps.
- **Full-Speed (USB 1.1)**: estima uma velocidade de 12Mbps.
- **High-Speed (USB 2.0)**: estima uma velocidade de 480Mbps.
- **SuperSpeed (USB 3.0)**: estima uma velocidade de 5.0Gbps.

A velocidade suportada pelo dispositivo é indicada pela linha de dado. Se o dispositivo opera no modo *Low-Speed*, então deve ser posto um resistor *pull-up* de $1K5\pm5\%$ na linha D-. Caso opere no modo *Full-Speed*, um resistor *pull-up* de $1K5\pm5\%$ deve ser inserido na linha D+.

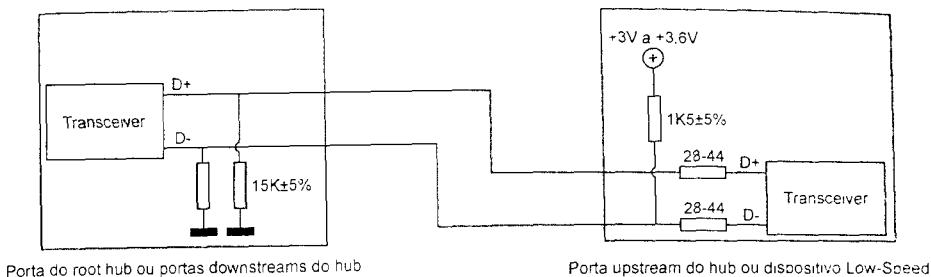
Dispositivos que funcionam no modo *High-Speed* adotam o mesmo tipo de conexão do *Full-Speed*, porém possuem um protocolo especial durante o *reset*, que verifica se o modo *High-Speed* é suportado pelo dispositivo.



Porta do root hub ou portas downstreams do hub

Porta upstream do hub ou dispositivo Full-Speed

Figura 18.4: Conexão dos resistores em dispositivos Full-Speed.

Figura 18.5: Conexão dos resistores em dispositivos *Low-Speed*.

18.5 Codificação/Decodificação NRZI

A transferência de pacotes em um barramento USB emprega a codificação/decodificação de dados NRZI (*Non Return to Zero Invert*), que consiste em definir '0' ou '1' pela mudança de nível de tensão nas linhas D+ e D-, com níveis opostos.

Em que '1' é representado quando não há mudança de nível e '0' é representado quando há mudança de nível. A Tabela 18.1B lista os possíveis estados.

Veja a seguir um exemplo de codificação de dado NRZI.

Tabela 18.1B. Nome dos estados.

Estado	Linha D+	Linha D-
J	1	0
K	0	1
SE0 (Single-Ended Zero)	0	0

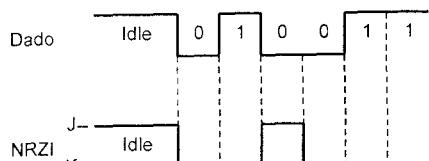


Figura 18.6: Codificação de dado NRZI.

Quando uma sequência de dois estados SE0 é verificada pelo dispositivo USB, ele interpreta como um sinal de *End-of-Packet* (EOP) e entra no modo *Idle* (modo de espera). O dispositivo sai do modo de espera ao verificar atividade no barramento, ou seja, sai do estado J e vai para o estado K, como pode ser observado na Figura 18.6. Esse processo é conhecido como *Start-of-Packet* (SOP) e sinaliza o início do pacote.

Se o *host* enviar um EOP, porém, e verificar que as duas linhas de dados permaneceram em nível baixo, ele vai interpretar que o dispositivo foi desconectado do barramento USB, pois não há o desbalanceamento das linhas D- e D+ ocasionado pelo resistor *pull-up* do dispositivo.



Se o *hub* ou *root hub* conduzir um SE0 na porta *downstream* por um período de tempo superior a 10ms, os dispositivos conectados a ele interpretarão como um sinal de **reset**. Para melhor compreensão, veja o processo de enumeração.

18.6 Endpoint e Pipe

Endpoint é o ponto final do fluxo de dados entre o *host* e o dispositivo, cujo sentido do dado pode ser configurado como entrada ou saída. Dispositivos *Low-Speed* podem possuir apenas três endpoints (0-2), enquanto dispositivos *Full-Speed* podem possuir até 16 endpoints (0-15). O endpoint 0 de qualquer dispositivo é implementado como controle, e através dele o *host* manipula e inicializa o dispositivo.

Um *pipe* (canal) é a associação entre um *endpoint* do periférico e o *software* do *host*. Ele faz a ligação entre o *buffer* do *host* e o *endpoint* do dispositivo, podendo um dispositivo USB possuir vários *pipes*. Os *pipes* também estão associados com largura de banda, tipo de serviço e características do *endpoint*, tais como a direção e o tamanho do *buffer*.

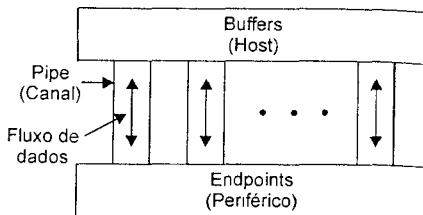


Figura 18.7: Endpoint e pipe.

Há dois tipos de *pipes*, sendo *stream pipe* e *message pipe*. O *stream pipe* não possui uma estrutura definida, ou seja, o conteúdo do dado não é interpretado pelo USB, enquanto a estrutura do *message pipe* é definida pelo protocolo, e a estrutura dos dados deve estar de acordo com as definições da USB.

Sempre que um dispositivo é conectado na interface USB, um *Default Control Pipe* (*endpoint 0 - message pipe*) é estabelecido com a finalidade de acessar configuração, *status* e informações de controle do dispositivo.

O USB envolve quatro tipos de transferência de dados. São eles:

- **Control Transfers:** usado para configurar os dispositivos quando são ligados ao barramento pela primeira vez, além de outras aplicações específicas, definidas pelo *software* de controle.
- **Bulk Data Transfers:** usado em transferências de grandes quantidades de dados de forma sequencial, como em impressoras, scanners, entre outros. Esse tipo de transferência difere da *Isochronous*, pois garante a entrega dos dados através de mecanismos de verificação de erro. No entanto, não garante a banda.
- **Interrupt Data Transfers:** usado por dispositivos que necessitam transmitir ou receber pequenas quantidades de dados a uma velocidade baixa, e de tempo em tempo, por exemplo, mouse e teclado. Esse tipo de transferência garante o tempo da entrega do pacote e a confiabilidade dos dados devido aos mecanismos de verificação de erro.
- **Isochronous Data Transfers:** usado em transferências de grandes quantidades de dados de forma sequencial, com largura de banda e latência de entrega pré-negociada, e tolerante a erros, como é observado em transmissões de áudio e vídeo. Essa transferência é a mais veloz e a menos confiável de todos os tipos de transferência, pois não retorna o pacote de *handshake*, que é essencial para verificação da correta entrega de dados.

18.7 Protocolo USB

Em um barramento USB, todos os pacotes são iniciados com um campo de sincronização (SYNC) que é enviado pelo *host* e utilizado pelo dispositivo para alinhar o dado de entrada com o *clock* local e indicar o início de um pacote. Esse campo possui tamanho de 8bits para o modo *Full-Speed/Low-Speed* e 32bits para o *High-Speed*, sendo os dois últimos bits, responsáveis pela sinalização do fim da sincronização e início do PID (*Packet Identifier*). Veja a seguir os modelos do campo de sincronização de acordo com o barramento, Figura 18.8.

	Low-Speed Full-Speed	High-Speed
NRZI	3 pares de KJ seguidos de KK	15 pares de KJ seguidos de KK
Dado	0000 0001	0000 0000 0000 0000 0000 0000 0000 0001

Figura 18.8: Campo de sincronização de acordo com o barramento.

Em uma comunicação USB, as transações ocorrem dentro de *frames* (quadros) ou *microframes* (micro-quadros). Um *frame* equivale a 1ms e está associado a um barramento *Low-Speed* ou *Full-Speed*, e um *microframe* equivale a 125us e está associado a um barramento *High-Speed*.

Cada *frame* ou *microframe* é iniciado com um *Start-of-frame* (SOF), que especifica o número do *frame* atual representado em 11bits, e gerado a cada 1ms para barramentos *Low-Speed* ou *Full-Speed* e, por padrão, a cada oito períodos de 125us para barramentos *High-Speed*, podendo ser alterado. Seu formato pode ser visto a seguir:

	SYNC	PID	Número do Frame	CRC5
Low-Speed Full-Speed	8bits	8bits	11bits	5bits
High-Speed	32bits	8bits	11bits	5bits

Figura 18.9: Formato do SOF.

O campo PID (*Packet Identifier*) possui tamanho de 8bits e especifica o tipo de pacote que será transferido, neste caso um SOF token. Veja a Tabela 18.2.

O campo *número do frame* possui tamanho de 11bits, cujo valor corresponde ao número do *frame/microframe* atual.

O campo CRC5 (*Cyclic Redundancy Checks*) dispõe a verificação de erro do campo *número do frame*.



O pacote SOF deve ser enviado somente no início de cada *frame/microframe*.

Em uma comunicação USB, o envio dos pacotes é iniciado pelo *bit* menos significativo (LSb) e finalizado com o *bit* mais significativo (MSb).

18.7.1 Campo de Identificação do Pacote

O campo PID (*Packet Identifier*) é sempre enviado após o processo de sincronização (SYNC) e indica o tipo de pacote que será transferido, podendo ser *token*, *data*, *handshake* ou *special*.

Esse campo é representado por 8bits, no entanto apenas os quatro primeiros *bits* representam o valor do código (Tabela 18.2 - Campo PID<3:0>) relacionado ao tipo de pacote, os outros 4bits (*Bit 4 - Bit 7*) são o complemento desses quatro primeiros *bits* (*Bit 0 - Bit 3*) e utilizados pelo receptor para verificação de erro.

Veja a seguir a lista dos tipos de PID e seus respectivos códigos.

bit 0	bit 3	bit 4	bit 7				
PID				PID			
0	1	2	3	0	1	2	3

Figura 18.10: Formato do PID.

Tabela 18.2: Lista dos tipos de PID.

Tipo do pacote	Nome do PID	Tipo de transferência	PID<3:0>	Descrição
Token	OUT	Todos	0001	Informa o endereço do dispositivo e o número do <i>endpoint</i> que será utilizado na transação <i>host</i> para periférico.
	IN	Todos	1001	Informa o endereço do dispositivo e o número do <i>endpoint</i> que será utilizado na transação periférico para <i>host</i> .
	SOF	Start-of-Frame	0101	Sinaliza o inicio do <i>frame/microframe</i> e o seu número.
	SETUP	Control	1101	Informa o endereço do dispositivo e o número do <i>endpoint</i> que será utilizado no processo de configuração.

Tipo do pacote	Nome do PID	Tipo de transferência	PID<3:0>	Descrição
Data	DATA0	Todos	0011	Usado para transferência de dados. Observação: Mecanismo de <i>data toggle</i> .
	DATA1	Todos	1011	Usado para transferência de dados. Observação: Mecanismo de <i>data toggle</i> .
	DATA2	<i>Isochronous</i>	0111	Usado para transferência de dados. Observação: Transferência <i>High-Speed isochronous</i> .
	MDATA	<i>Isochronous e Interrupt</i>	1111	Usado para transferência de dados. Observação: Transferência <i>High-Speed isochronous</i> .
Handshake	ACK	Todos	0010	Sinaliza que o receptor aceitou o pacote.
	NAK	<i>Control, Bulk e Interrupt</i>	1010	Sinaliza que o receptor estava ocupado ou o dado foi recusado. Observação: Enviado apenas pelos dispositivos USB
	STALL	<i>Control, Bulk e Interrupt</i>	1110	Sinaliza que a solicitação de controle não era suportada, houve uma falha na requisição de controle ou ocorreu uma falha de <i>endpoint</i> . Observação: Enviado apenas pelos dispositivos USB
	NYET	<i>Control Write, Bulk OUT e transação Split</i>	0110	Sinaliza que o receptor ainda não está pronto. Observação: Enviado apenas pelos dispositivos USB
Special	PRE	<i>Control e Interrupt</i>	1100	É um pacote do tipo <i>token</i> , utilizado para habilitar a porta <i>downstream</i> do <i>host</i> para se comunicar com dispositivos <i>Low-Speed</i> .
	ERR	Todos	1100	É um pacote do tipo <i>handshake</i> , utilizado para informar erro na transação <i>Split</i> . Usado apenas em <i>High-Speed hub</i> .
	SPLIT	Todos	1000	É um pacote do tipo <i>token</i> , utilizado em transações entre o <i>host</i> e o <i>High-speed Hub</i> com dispositivos <i>Low-Speed</i> e <i>High-Speed</i> conectados.
	PING	<i>Control Write e Bulk OUT</i>	0100	É um pacote do tipo <i>token</i> , utilizado para conferir se o <i>endpoint</i> está ocupado. Ele auxilia no controle de fluxo <i>High-speed</i> para um <i>bulk/control endpoint</i> .
	Reservado	-	0000	Reservado.



Observe que o campo PID<3:0> representa o código relacionado ao tipo de pacote do bit mais significativo para o menos significativo, porém a sequência dos bits do campo PID apresentado na Figura 18.10 é inversa.

Se o PID for válido, mas não for suportado pelo dispositivo, ele não responde.

18.7.2 Pacote Token

Toda transação é controlada pelo *host* e, na maioria dos casos, envolve três pacotes responsáveis por descrever o tipo e a direção da transação (campo PID), o endereço do periférico (campo ADDR) e o número do *endpoint* (campo ENDP). Esse pacote é conhecido como *Token Packet*.

	SYNC	PID	ADDR	ENDP	CRC5
Low-Speed Full-Speed	8bits	8bits	7bits	4bits	5bits
High-Speed	32bits	8bits	7bits	4bits	5bits

Figura 18.11: Formato do *Token Packet*.

Uma vez estabelecida a conexão, os dados podem trafegar do *host* para o periférico ou vice-versa, sendo a direção especificada pelo *Token Packet*.

18.7.2.1 Campo ADDR

O campo ADDR informa o endereço do dispositivo, ao qual o pacote é destinado. Esse endereço é representado por 7bits e especificado pelo *host* no processo de enumeração, que ocorre quando o dispositivo é conectado no barramento pela primeira vez.

18.7.2.2 Campo ENDP

O campo ENDP é composto de 4bits e especifica o endpoint (*endpoint* 0 a 15) do dispositivo, o qual será usado para transmitir (transação IN) ou receber dados (transação OUT ou SETUP).

18.7.2.3 Campo CRC5

O campo CRC5 (*Cyclic Redundancy Checks*) é composto de 5bits e contém o valor calculado dos campos ADDR e ENDP. O valor desse campo é automaticamente calculado pela SIE do receptor, como forma de verificação de erros no pacote.

18.7.3 Pacote Data

Como mencionado anteriormente, o *host* é responsável pelo gerenciamento do barramento USB e os dispositivos devem responder às suas solicitações.

O *host* pode enviar dados ou solicitar o envio de dados para o dispositivo. O envio de dados do *host* para o dispositivo é feito pelo envio de um OUT ou SETUP *token*, Tabela 18.2. Já a solicitação de envio de dados do dispositivo para o *host* ocorre pelo envio de um IN *token*, Tabela 18.2, endereçado ao endpoint do dispositivo, o qual envia um pacote de dado como resposta à solicitação.

Existem quatro tipos de pacotes de dados, DATA0, DATA1, DATA2 e MDATA_n, todos constituídos de um campo PID, dado (0-8bytes (*Low-Speed*), 0-1023bytes (*Full-Speed*) e 0-1024bytes (*High-Speed*)) e por último um campo CRC16, cujo cálculo não leva em consideração o campo PID, apenas os dados transferidos.

	SYNC	PID	DATA	CRC16
Low-Speed	8bits	8bits	0 - 64bits	16bits
Full-Speed	8bits	8bits	0 - 8184bits	16bits
High-Speed	32bits	8bits	0 - 8192bits	16bits

Figura 18.12: Formato do pacote de dados.

A USB dispõe de um mecanismo chamado *data toggle*, utilizado para garantir a sincronização de uma sequência de dados entre o transmissor e o receptor. Esse mecanismo utiliza o pacote DATA0 e o DATA1 e consiste em alternar os pacotes para envio/recepção de acordo com o valor do *data toggle bit* ("bit de sequência"), o qual é utilizado para identificar que pacote deve ser usado na transação.

Seu funcionamento é simples. Quando o receptor recebe um pacote de dados válidos e o seu *bit* de sequência corresponde ao pacote PID (DATA0 ou DATA1), o *bit* de sequência é alternado no receptor e um pacote ACK *handshake* é enviado ao transmissor, informando que a recepção do pacote foi bem-sucedida.

O transmissor, ao receber o ACK *handshake*, alterna o seu *bit* de sequência e assim por diante. Veja o exemplo na Figura 18.13.

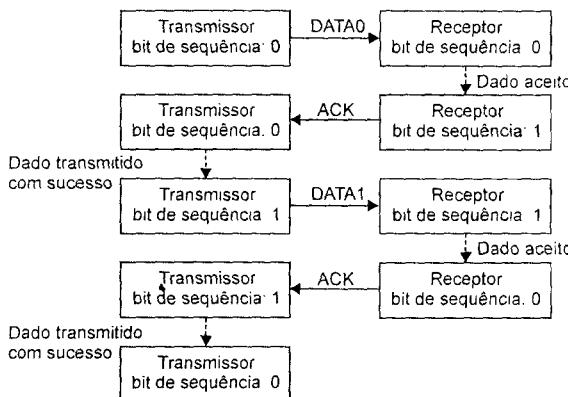


Figura 18.13: Transação sequencial de dados sem erro.

Se o receptor não aceitar o pacote de dado, seu *bit de sequência* permanece inalterado e envia um NAK *handshake* para o transmissor, que também mantém o valor de seu *bit de sequência*.

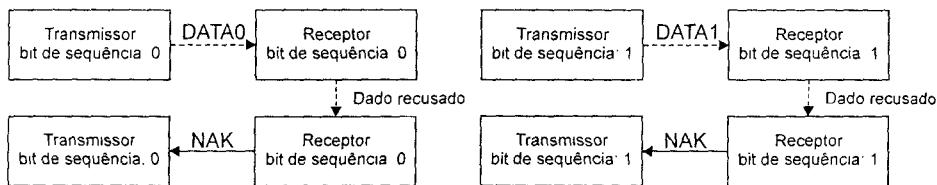


Figura 18.14: Transação sequencial de dados com erro.

Outros problemas também podem ocorrer, tais como falha no recebimento de um ACK *handshake* e *bit de sequência* não compatível.

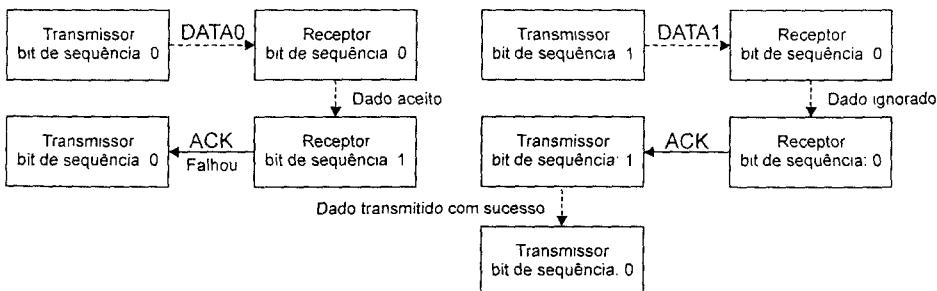


Figura 18.15: Transação sequencial de dados com erro de handshake e bit de sequência.



Se o tipo de transferência do *endpoint* for *Bulk*, *Control* ou *Interrupt*, o receptor responde com um pacote de ACK *handshake* ao receber o dado sem erro. No entanto, se for do tipo *Isochronous*, não há uma resposta.

18.7.4 Pacote de Handshake

O pacote de *Handshake* é usado para informar o *status* de uma transação que suporta controle de fluxo. Esse pacote consiste em apenas um PID e seu formato pode ser visto na Figura 18.16.

Os seguintes estados podem ser retornados pelos pacotes de *Handshake*:

- **ACK:** indica que o pacote de dados foi recebido sem erro. O ACK *handshake* pode ser retornado pelo *host* ou pelo dispositivo, dependendo do tipo de transação. Se for uma transação IN, então o *host* envia o pacote ACK *handshake*, e se for uma transação OUT/SETUP/PING, o dispositivo envia esse pacote.
- **NAK:** indica que o dispositivo está ocupado ou ocorreu um erro de CRC. Esse tipo de pacote deve ser retornado apenas pelo dispositivo, em uma transação IN (fase de dado) ou transação OUT/PING (fase de *handshake*).
- **STALL:** indica que o dispositivo não está habilitado para receber/transmitir dados ou o *pipe* de controle solicitado não é suportado pelo dispositivo. Esse pacote é retornado pelo dispositivo, em resposta a um IN *token* ou depois de uma fase de dados de uma transação OUT, ou como resposta a uma transação PING.
- **NYET:** esse pacote é retornado por um High-Speed endpoint como parte do protocolo PING, como também pode ser retornado por um hub em uma transação Split, no caso de não suportar a transação Split ou quando a transação Full-Speed/Low-Speed ainda estiver em andamento.
- **ERR:** indica a ocorrência de erro em uma transação Split.



O *host* pode enviar apenas o ACK *handshake*.

18.7.4.1 Transação IN

Uma transação IN pode envolver dois pacotes de *handshake*, um enviado pelo dispositivo e outro pelo *host*. Veja a seguir as possíveis condições.

- **Transação IN:** resposta do dispositivo.
 - Se o IN *token* for corrompido, o dispositivo simplesmente não responde.
 - Se o dispositivo receber o IN *token* e seu recurso *Halt* estiver selecionado, é retornado um STALL *handshake*.
 - Se o dispositivo receber o IN *token* e seu recurso *Halt* não estiver selecionado, mas o dispositivo não tiver dado para enviar, é retornado um NAK *handshake*.
 - Se o dispositivo receber o IN *token* e seu recurso *Halt* não estiver selecionado, e o dispositivo tiver dado para enviar, é retornado o pacote de dados.
- **Transação IN:** resposta do *host*.
 - Se o pacote de dados recebido pelo *host* estiver corrompido, o *host* descarta os dados e não responde.
 - Se o pacote de dados for corretamente recebido, porém não puder ser aceito pelo *host*, então ele descarta os dados e não responde.
 - Se o pacote de dados for corretamente recebido e aceito pelo *host*, ele responde com um ACK *handshake*.

	SYNC	PID
Low-Speed Full-Speed	8bits	8bits
High-Speed	32bits	8bits

Figura 18.16: Formato do pacote de *Handshake*

18.7.4.2 Transação OUT

Em uma transação OUT apenas o dispositivo responde com um pacote de *Handshake*. As possíveis respostas estão listadas a seguir.

- Se o pacote de dados recebido pelo dispositivo estiver corrompido, o dispositivo não responde.
- Se o dispositivo receber o pacote de dados sem erros, porém estiver com o recurso *Halt* selecionado, ele retorna um STALL *handshake*.
- Se o dispositivo receber o pacote de dados sem erros, não estiver com o recurso *Halt* selecionado, mas o *bit* de sequência não corresponde ao seu, então é enviado um ACK *handshake* de modo a permitir que o *host* e o dispositivo voltem a estar sincronizados.
- Se o dispositivo receber o pacote de dados sem erros, não estiver com o recurso *Halt* selecionado, com *bit* de sequência correto, mas o dado não puder ser aceito pelo dispositivo, é enviado um NAK *handshake*.
- Se o dispositivo receber o pacote de dados sem erros, não estiver com o recurso *Halt* selecionado, com *bit* de sequência correto, e o dado puder ser aceito pelo dispositivo, então é enviado um ACK *handshake*.

18.7.4.3 Transação SETUP

Quando o dispositivo recebe um SETUP *token*, ele não pode responder com STALL nem NAK e deve aceitar o pacote seguido do SETUP *token*.



Se um *endpoint* sem controle receber um SETUP *token*, ele deve ignorar a transação e não responder.

18.7.5 Pacote Special

O pacote *special preamble* (PRE) é enviado pelo *host* quando ele deseja comunicar-se com dispositivos *Low-Speed*. Digamos que um *hub* esteja se comunicando com dispositivos *Low-Speed* e *Full-Speed*. Logo, temos dois sinais trafegando em velocidades diferentes. Para evitar que um dispositivo *Low-Speed* interprete o dado de forma incorreta, quando ele está sendo enviado a *high speed*, o *hub* desativa as portas *downstreams* em que dispositivos *Low-Speed* estão conectados. Para que o *hub* possa se comunicar com um dispositivo *Low-Speed*, o *host* deve enviar um pacote PRE em *Full-Speed*, que será interpretado pelo *hub* e ignorado por outros dispositivos, e após um período de configuração o *hub* deve estar preparado para repassar o sinal *low speed* para as portas *downstreams* onde dispositivos *Low-Speed* estão conectados.

O *special SPLIT token* é um pacote com 4bytes. Esse tipo de transação é usado para dar suporte às transações de *split* entre o *host* que se comunica com um *High-Speed hub* que possui dispositivos *Full-Speed/Low-Speed* conectados a ele.

O pacote *special PING token* é enviado pelo *host* para verificar se um *endpoint* de um dispositivo *High-Speed* está ocupado, antes de enviar o próximo pacote de dado em um OUT Control/Bulk transfer com múltiplos pacotes de dados.

18.8 Funcionamento dos Tipos de Transferência

Vamos estudar as características e o funcionamento dos tipos de transferência suportados pelo protocolo USB 2.0.



18.8.1 Bulk Data Transfers

A *Bulk Data Transfers* é utilizada quando se deseja transferir grandes volumes de dados de modo sequencial e com confiabilidade na entrega dos dados. Os dados podem trafegar do *host* para o dispositivo (*OUT token*), ou os dados podem trafegar do dispositivo para o *host* (*IN token*). A principal diferença dessa transferência com relação à isócrona (*isochronous*) é que, após a etapa de dados, há uma fase de *handshake* a qual assegura que o dado está sendo transferido corretamente. Veja a seguir o formato da transferência *Bulk*.

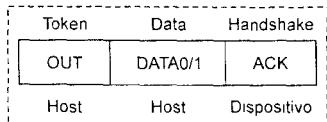


Figura 18.17: Host transmite dados.

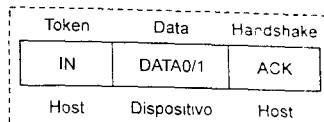


Figura 18.18: Host recebe de dados

O dispositivo pode se comportar de três formas distintas, quando recebe um IN token:

- Retorna o dado.
- Retorna um NAK, caso esteja ocupado.
- Retorna um STALL, caso o endpoint indicado apresente algum erro.

O pacote de *handshake* pode apresentar um dos três estados listados a seguir.

- **Sucesso:** ACK handshake.
- **Ocupado:** NAK handshake (enviado apenas pelo dispositivo).
- **Falha:** STALL handshake (enviado apenas pelo dispositivo).

O tamanho do pacote para dispositivos *Full-Speed* é 8, 16, 32 ou 64bytes, enquanto para dispositivos *High-Speed* é 512bytes.



Esse tipo de transação não é suportado por dispositivos *Low-Speed*.

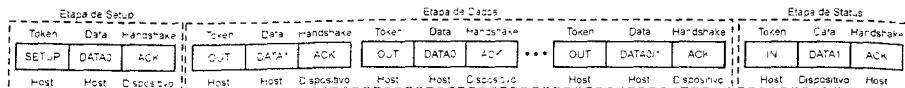
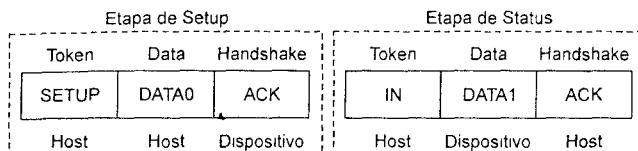
18.8.2 Control Transfers

As *Control Transfers* são destinadas à configuração/comando/status da comunicação entre o *host* e dispositivos USB.

A transferência inicia-se com a etapa de *Setup*, em que o *host* fornece informação sobre a solicitação para o periférico, em seguida vem a etapa de *Data* (opcional), cuja direção de transferência dos dados é informada pelo estágio de *Setup*, e por último, o dispositivo USB retorna um *handshake*, na etapa de *Status*. Veja a seguir as possíveis transações de *Setup*.



Figura 18.19: Transação de *Setup* com recepção de dados.

Figura 18.20: Transação de *Setup* com transmissão de dados.Figura 18.21: Transação de *Setup* sem transferência de dados.

18.8.2.1 Etapa de Setup

Na etapa de **Setup**, o *host* envia um SETUP token seguido de um pacote DATA0 com 8bytes de dados que especificam o tipo de solicitação que o *host* deseja que o dispositivo USB responda. Seu formato pode ser visto na Figura 18.22.

Byte 0	Byte 1	Byte 2-3	Byte 4-5	Byte 6-7
bmRequestType	bRequest	wValue	wIndex	wLength

Figura 18.22: Formato do pacote DATA0 para a etapa de *Setup*.

O primeiro byte (*bmRequestType*) do pacote DATA0 identifica o tipo de solicitação e está dividido em três campos, cada qual com sua função, como pode ser observado a seguir.

bit 7	bit 6	bit 5	bit 4	bit 0
Direction	Request Type	Recipient		

Figura 18.23: Formato do campo *bmRequestType*.

O bit *Direction* especifica a direção dos pacotes na etapa de **Data**. Se for '1', os pacotes são transferidos do periférico para o *host* (IN); caso contrário, é do *host* para o periférico (OUT).

O bit *Request Type* define o tipo de solicitação, cuja combinação dos dois bits resulta em um dos elementos listados na Tabela 18.3.

Tabela 18.3: Lista dos tipos de solicitação.

Request Type	Descrição
00	Solicitação de uma das 11 requisições padrão.
01	Solicitação definida por uma classe específica.
10	Solicitação definida por um fornecedor.

Veja na Tabela 18.4 as 11 requisições padrão.



Tabela 18.4: Lista das requisições padrão.

bRequest	Nome	Recipient (Receptor)	wValue	wIndex	wLength (Etapa de dados)
0x00	Get_Status	Dispositivo Interface Endpoint	0	Dispositivo Interface Endpoint	2
0x01	Clear_Feature	Dispositivo Interface Endpoint	Recurso	Dispositivo Interface Endpoint	0
0x03	Set_Feature	Dispositivo Interface Endpoint	Recurso	Dispositivo Interface Endpoint	0
0x05	Set_Address	Dispositivo	Endereço do dispositivo	0	0
0x06	Get_Descriptor	Dispositivo	Índice e tipo do descritor	Dispositivo ou identificador de idioma	Tamanho do descritor
0x07	Set_Descriptor	Dispositivo	Índice e tipo do descritor	Dispositivo ou identificador de idioma	Tamanho do descritor
0x08	Get_Configuration	Dispositivo	0	Dispositivo	1
0x09	Set_Configuration	Dispositivo	Configuração	Dispositivo	0
0x0A	Get_Interface	Interface	0	Interface	1
0x0B	Set_Interface	Interface	Interface	Interface	0
0x0C	Synch_Frame	Endpoint	0	Endpoint	2

O bit *Recipient* especifica quem vai receber o pacote. Veja a seguir a lista dos receptores.

Tabela 18.5: Lista dos receptores.

Recipient	Descrição
0000	Solicitação direcionada para o dispositivo.
0001	Solicitação direcionada para uma interface específica.
0010	Solicitação direcionada para o endpoint.
0011	Solicitação direcionada para outro elemento dentro do dispositivo.

O segundo byte (*bRequest*) especifica a requisição e está diretamente relacionado ao *bit Request Type*. Se o *bit Request Type* for igual a '00', o *bRequest* vai conter uma das 11 requisições padrão, de acordo com a Tabela 18.4. Se for '01', o *bRequest* nomeia uma requisição definida pela classe do dispositivo, e por último, se o *bit Request Type* for igual a '10', o *bRequest* nomeia uma requisição definida pelo fornecedor do dispositivo.

O campo *wValue* é composto de 2bytes. O significado desses bytes pode ser definido de acordo com a solicitação, por exemplo, em uma solicitação *Set_Address*, eles contêm o endereço do dispositivo.

O campo *wIndex* também é composto de 2bytes, e do mesmo modo que o campo *wValue*, seu significado depende diretamente da solicitação. Esse campo é normalmente usado para passar o número do *endpoint* ou interface. Quando usado para indexar um *endpoint*, os 4bits menos significativos definem o número do *endpoint* e o bit 7 especifica a direção do *endpoint*, sendo '1' para um IN e '0' para um OUT. Se esse campo for usado para indexar uma interface, os 8bits menos significativos representam o número da interface.

O campo *wLength* informa o comprimento do pacote que será transferido no estágio de *Data*. Se a direção for host → periférico, então o valor desse campo indica o número de bytes que o host vai transferir, porém se a direção for oposta, esse valor representa o número máximo de bytes que o dispositivo vai transmitir, podendo este transferir uma quantidade menor a que foi especificada. Caso o valor desse campo seja zero, indica que o estágio de *Data* não existe.

Após receber o pacote DATA0, o periférico USB responde com um ACK handshake, caso não tenha verificado erros na recepção; caso contrário, retorna um erro.

18.8.2.2 Etapa de Data

O próximo estágio é o **Data**. Nesse estágio, o *host* envia um IN/OUT token para o dispositivo USB, de acordo com a direção especificada pela etapa de **Setup**, e recebe/transmite uma sequência de pacotes de dados, alternando entre DATA0 e DATA1. Se o *host* recebeu corretamente uma sequência de pacotes de dados do periférico, ele retorna um ACK *handshake*; caso contrário, reporta um erro. Se o *host* transmitiu uma sequência de pacotes de dados para o periférico, este último retorna um ACK *handshake*; caso contrário, reporta um erro.

18.8.2.3 Etapa de Status

O último estágio é o **Status**. Ele depende diretamente da direção definida no estágio de **Setup**. Se for um IN, então o *host* envia um OUT token; caso contrário, envia um IN token. Em seguida é transferido um pacote DATA1 nulo, e após receber esse pacote, o *host/periférico* retorna um ACK *handshake*, caso não sejam verificados erros; senão, retorna um erro. As três condições para a etapa **Status** são:

- A sequência de comando foi completada corretamente.
- A sequência de comando falhou.
- O dispositivo está ocupado.

Em uma transferência de escrita de controle, após o dispositivo receber os dados na etapa de **Data**, ele retorna o status da transferência através da etapa de **Status**. Se o dispositivo retornar um pacote de dados nulo, significa que a transferência foi bem-sucedida. Em contrapartida, se um NAK ou STALL for retornado, significa que houve problema na transferência. O *host* retorna um ACK *handshake* com a finalidade de indicar que recebeu o status da etapa de **Status**.

Tabela 18.6: Escrita de controle.

Escrita de controle - Host envia dados para o dispositivo		
Direção da etapa de Status	Pacote de dados da etapa de Status	Pacote de handshake da etapa de Status
IN	Sucesso - pacote de dados nulo. Ocupado - NAK. Falha - STALL	Host retorna ACK

Em uma transferência de leitura de controle, após o dispositivo enviar os dados na etapa de **Data**, o *host* retorna um ACK *handshake* com a finalidade de indicar que recebeu o status da etapa de **Status**; caso contrário, não responde. Nesse tipo de transferência, o *host* aguarda a chegada de todos os pacotes de dados da etapa de **Data**, em seguida solicita o status da transferência através da etapa de **Status**. Se o pacote de dados enviado pelo dispositivo for nulo, indica que houve sucesso na transferência; caso contrário, significa que ocorreu um problema na transferência (NAK ou STALL).

Tabela 18.7: Leitura de controle.

Leitura de controle - Host recebe dados do dispositivo		
Direção da etapa de Status	Pacote de dados da etapa de Status	Pacote de handshake da etapa de Status
OUT	Sucesso - pacote de dados nulo.	Sucesso - ACK. Ocupado - NAK. Falha - STALL.

18.8.3 Isochronous Data Transfers

Esse tipo de transferência não possui controle de erro nem pacote de *handshake*, resultando em uma transferência de dados com taxas elevadas. Esse tipo de transferência é bidirecional, ou seja, se o *host* enviar um OUT token, os dados vão trafegar do *host* para o dispositivo; caso contrário, se o *host* enviar um IN token, os dados vão trafegar do dispositivo para o *host*.

A transferência High-Speed isochronous IN endpoint suporta múltiplas transações em um mesmo *microframe*, podendo ser configurada como uma transação por *microframe*, duas transações por *microframe* ou três transações por *microframe*. Veja a seguir a sequência dos pacotes de dados para os três modos de configuração.

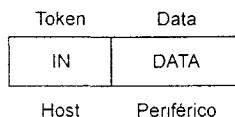


Figura 18.24: Recepção de dados - uma transação por *microframe*.

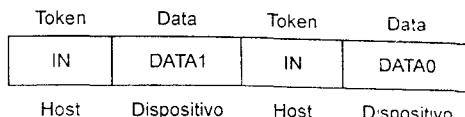


Figura 18.25: Recepção de dados - duas transações por *microframe*.

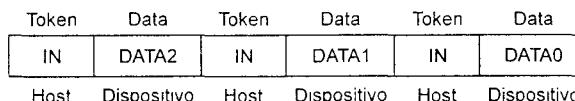


Figura 18.26: Recepção de dados - três transações por *microframe*.

A transferência High-Speed isochronous OUT endpoint possui uma sequência diferente da High-Speed isochronous IN. Acompanhe.

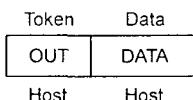


Figura 18.27: Transmissão de dados - uma transação por *microframe*.

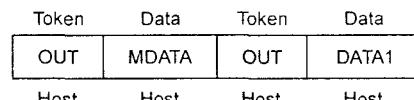


Figura 18.28: Transmissão de dados - duas transações por *microframe*.

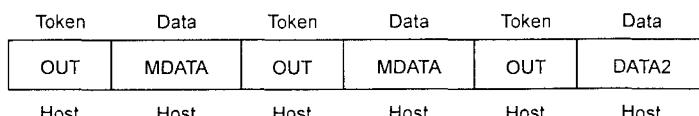


Figura 18.29: Transmissão de dados - três transações por *microframe*.

A Tabela 18.8 lista a quantidade de bytes que podem ser transmitidos por cada pacote de acordo com as três configurações possíveis.

Tabela 18.8: Número de transações por *microframe*.

Número de transações por <i>microframe</i>	Tamanho máximo de bytes por pacote
1	1 a 1024
2	513 a 1024
3	683 a 1024



A transferência *High-Speed isochronous* é capaz de transferir até 3072bytes (3 x 1024bytes) por microframe.

18.3.4 Interrupt Data Transfers

O *Interrupt Data Transfers* é similar a *Bulk Data Transfers*. A diferença é que o dispositivo possui um tempo de latência pré-negociado com o *host*, desta forma a transferência de dados ocorre periodicamente e a uma taxa de comunicação baixa.

Se o dispositivo receber um IN token, ele retorna um dado ou envia um pacote NAK ou STALL, reportando erro. Se o dispositivo não tiver nenhum dado para enviar, retorna um NAK handshake. Se uma falha for verificada, o dispositivo retorna um STALL handshake. Após receber o pacote de dados, o *host* envia um ACK handshake informando que a transação foi bem-sucedida; caso contrário, simplesmente não responde, Figura 18.30.

Se o dispositivo receber um OUT token, o *host* envia um dado para o dispositivo. Se o dispositivo receber o dado sem a ocorrência de erros, então retorna um ACK handshake. Se o dispositivo estiver ocupado, ele retorna um NAK, e se estiver em uma condição de erro no endpoint, retorna um STALL, Figura 18.31.

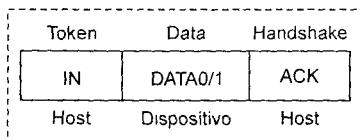


Figura 18.30: Recepção de dados.

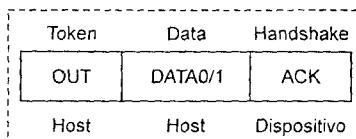


Figura 18.31: Transmissão de dados.

Para dispositivos *Low-Speed*, o tamanho do pacote é de 1 a 8bytes e o tamanho do pacote para dispositivos *Full-Speed* é de 1 a 64bytes, enquanto para dispositivos *High-Speed* é de 1 a 1024bytes.

18.9 Recursos Padrão do Dispositivo USB

Existem três recursos padrão que podem ser suportados ou não pelos dispositivos USB. São eles: *device remote wakeup*, *endpoint halt* e *test mode*.

18.9.1 Device Remote Wakeup

O *device remote wakeup* é um recurso do dispositivo que permite a ele gerar um evento que solicita ao *host* sair do estado suspenso e voltar à condição de ativo. Ele pode ser alterado pelas solicitações SET_FEATURE ou CLEAR_FEATURE, abordadas mais adiante.

18.9.2 Endpoint Halt

O recurso *endpoint halt* permite ao dispositivo colocar um determinado endpoint em uma condição *halted* ("Interrompido") quando é detectado um problema na comunicação, por exemplo, quando o dispositivo está esperando um pacote de handshake e ele não é recebido ou se a quantidade de dados recebidos excedeu ao esperado. Outro evento que pode ser utilizado para gerar uma condição de *halt* é o envio de uma solicitação SET_FEATURE com argumento ENDPOINT_HALT.

A condição de *halt* pode ser removida pelo envio da solicitação CLEAR_FEATURE com o argumento ENDPOINT_HALT.



Esse recurso é necessário para todo tipo de *Interrupt* e *Bulk endpoint*.

18.9.3 Test Mode

O recurso *test mode* deve ser suportado por todos os *host controllers*, *hubs* e dispositivos *High-Speed*. Ele pode ser executado por meio da solicitação *SET_FEATURE* com o argumento *TEST_MODE*. Os modos de teste podem ser vistos na Tabela 18.9.

Tabela 18.9: Modos de teste.

Valor	Descrição
0x00	Reservado
0x01	Test_J
0x02	Test_K
0x03	Test_SE0_NAK
0x04	Test_Packet
0x05	Test_Force_Enable
0x06 - 0x3F	Reservado para futuros testes padrão
0x3F - 0xBF	Reservado
0xC0 - 0xFF	Reservado para modos de teste do fornecedor

A transação para o modo de teste deve ser realizada em um período de tempo inferior a 3ms após concluir a etapa de *Status* da solicitação.

A seguir, temos de modo resumido o significado de cada um dos modos de teste.

18.9.3.1 TEST_J

A porta do *transceiver* é colocada e mantida no estado J até que o dispositivo seja desligado ou o *hub* reiniciado.

18.9.3.2 TEST_K

A porta do *transceiver* é colocada e mantida no estado K até que o dispositivo seja desligado ou o *hub* reiniciado.

18.9.3.3 TEST_SE0_NAK

A porta do *transceiver* entra e permanece no modo de recepção *High-Speed* até que uma ação de saída seja executada.

18.9.3.4 TEST_PACKET

O *transceiver* envia um pacote de teste padrão repetidamente até que uma ação de saída seja executada.

18.9.3.5 TEST_FORCE_ENABLE

As portas *downstream* do *hub* entram no modo *High-Speed* mesmo que não haja dispositivos conectados a elas, e os pacotes que entram na porta *upstream* do *hub* não devem ser encaminhados para a porta *downstream* que está sendo testada.



A porta *upstream* sai do modo de teste quando o dispositivo é desligado e alimentado novamente, e a porta *downstream* sai desse modo quando o *hub* é reiniciado.

18.10 Descritores Padrão

Para que o *host* possa estabelecer uma comunicação com um dispositivo, primeiramente deve conhecê-lo. Isso ocorre durante um processo chamado de enumeração, estudado mais adiante, em que descritores são solicitados pelo *host*, cujas informações retornadas permitem que ele aprenda sobre um determinado dispositivo e possa comunicar-se com ele.

Veja na Tabela 18.10 os tipos de descritor.

Tabela 18.10: Tipos de descritor.

Tipo de descritor	Valor
Device	1
Configuration	2
String	3
Interface	4
Endpoint	5
Device_qualifier	6
Other_speed_configuration	7
Interface_power	8

18.10.1 Configuration Descriptor

Cada dispositivo pode conter uma ou mais configurações. Cada configuração deve conter pelo menos uma interface e cada interface pode conter 0 ou mais *endpoints*. Isso implica que, se um descritor de configuração for solicitado pelo *host* (GET_CONFIGURATION), o dispositivo também retorne todos os descritores relacionados a essa configuração, tais como descritores das interfaces, *endpoints* e específicos da classe e do fabricante.

As configurações do dispositivo podem ser selecionadas pelo envio da seguinte solicitação SET_CONFIGURATION.

Tabela 18.11: Formato do configuration descriptor.

Offset	Campo	Tamanho	Valor	Descrição
0	bLength	1byte	Tamanho	Tamanho do descritor.
1	bDescriptorType	1byte	0x02	Tipo do descritor.
2	wTotalLength	2bytes	Quantidade	Quantidade total de dados retornados por essa configuração. Observação: Deve incluir os descritores das interfaces, <i>endpoints</i> e específicos da classe e do fabricante.
4	bNumInterfaces	1byte	Quantidade	Quantidade de interfaces suportadas por essa configuração. Observação: A configuração deve suportar pelo menos uma interface.
5	bConfigurationValue	1byte	Número	Valor usado como argumento para a solicitação SET_CONFIGURATION para selecionar essa configuração.
6	iConfiguration	1byte	Índice	Índice do descritor de string, descreve sua configuração.
7	bmAttributes	1byte	Bitmap	Atributos de configuração.
8	bMaxPower	1byte	mA	Máximo consumo de corrente pelo dispositivo. Cada unidade equivale a 2mA.

O campo *bmAttributes* informa as características de configuração do dispositivo, sendo representado por 1byte, cujos campos são:

Tabela 18.12: Formato do campo *bmAttributes*.

<i>Bit 7</i>	<i>Bit 6</i>	<i>Bit 5</i>	<i>Bit 4 a Bit 0</i>
Reservado	<i>Self-powered</i>	Remote Wakeup	Reservado
1	0 - Alimentado pelo barramento. 1 - Alimentação própria.	0 - Não suporta o recurso <i>Remote Wakeup</i> 1 - Suporta o recurso <i>Remote Wakeup</i> .	00000

Se o dispositivo estiver sendo alimentado por uma fonte externa, o valor do *bit 6* será '1' e o campo *bMaxPower* do descritor de configuração será igual a 0x00.

18.10.2 Device Descriptor

O *device descriptor* fornece informações gerais sobre o dispositivo, e cada dispositivo USB não pode apresentar mais de um descritor de dispositivo.

Tabela 18.13: Formato do *device descriptor*.

Offset	Campo	Tamanho	Valor	Descrição
0	<i>bLength</i>	1byte	Tamanho	Tamanho do descritor.
1	<i>bDescriptorType</i>	1byte	0x01	Tipo do descritor.
2	<i>bcdUSB</i>	2bytes	BCD	Número da versão, codificado em BCD. Seu formato é 0xJJMN, que corresponde à versão JJ.M N. Logo, temos que para um versão 2.0, o valor do campo <i>bcdUSB</i> é 0x0200
4	<i>bDeviceClass</i>	1byte	Classe	Código da classe.
5	<i>bDeviceSubClass</i>	1byte	Subclasse	Código da subclasse.
6	<i>bDeviceProtocol</i>	1byte	Protocolo	Código do protocolo.
7	<i>bMaxPacketSize0</i>	1byte	Quantidade	Tamanho máximo do pacote para o endpoint 0. Apenas os tamanhos de 8, 16, 32 e 64 estão disponíveis. Observação: Se o dispositivo estiver operando no modo <i>high-speed</i> , o tamanho do pacote deve ser 64bytes.
8	<i>idVendor</i>	2bytes	Identificação	Identificação do fabricante. (VID)
10	<i>idProduct</i>	2bytes	Identificação	Identificação do produto. (PID)
12	<i>bcdDevice</i>	2bytes	BCD	Versão do dispositivo, codificado em BCD.
14	<i>iManufacturer</i>	1byte	Índice	Índice do descritor de <i>string</i> , descrevendo o fabricante.
15	<i>iProduct</i>	1byte	Índice	Índice do descritor de <i>string</i> , descrevendo o produto.
16	<i>iSerialNumber</i>	1byte	Índice	Índice do descritor de <i>string</i> , descrevendo o número serial do dispositivo.
17	<i>bNumConfigurations</i>	1byte	Número	Número de configurações possíveis.

Os campos *bDeviceClass*, *bDeviceSubClass* e *bDeviceProtocol* são complementares. Veremos a seguir o que cada um representa.

- **bDeviceClass:** especifica a classe do dispositivo.
 - 0x00: indica que a informação da classe deve ser determinada pelos descritores de interface do dispositivo. Neste caso, o valor dos campos *bDeviceSubClass* e *bDeviceProtocol* também será igual a 0x00.
 - 0x01 a 0xFE: indica que o dispositivo suporta diferentes especificações de classe.
 - 0xFF: indica que a classe do dispositivo é especificada pelo fabricante.
- **bDeviceSubClass:** especifica a subclasse do dispositivo.
 - 0x00: esse campo é setado em 0x00, se o campo *bDeviceClass* for 0x00.
 - 0x01 a 0xFE: identifica a subclasse do dispositivo conforme a sua classe (*bDeviceClass*).
 - 0xFF: o valor da subclasse não é setado em 0xFF.
- **bDeviceProtocol:** especifica o protocolo usado pelo dispositivo com base em sua classe.
 - 0x00: indica que o dispositivo não usa um protocolo especificado pela classe.
 - 0x01 a 0xFE: indica que o dispositivo usa um protocolo especificado pela classe.
 - 0xFF: indica que o dispositivo usa um protocolo especificado pelo fabricante.



Todos os códigos dos campos *bDeviceClass*, *bDeviceSubClass* e *bDeviceProtocol* são definidos pela USB-IF e podem ser observados com maiores detalhes no site www.usb.org.

18.10.3 Device_Qualifier Descriptor

O *device_qualifier descriptor* fornece informações de como um dispositivo que opera no modo *High-Speed* vai operar no modo *Full-Speed* e vice-versa.

Tabela 18.14: Formato do device_qualifier descriptor.

Offset	Campo	Tamanho	Valor	Descrição
0	<i>bLength</i>	1byte	Tamanho	Tamanho do descritor.
1	<i>bDescriptorType</i>	1byte	0x06	Tipo do descritor.
2	<i>bcdUSB</i>	2bytes	BCD	Número da versão, codificado em BCD. Seu formato é 0xJJMN, que corresponde à versão JJ.M.N. Logo, temos que para um versão 2.0, o valor do campo <i>bcdUSB</i> é 0x0200. <i>Observação:</i> A versão deve ser igual ou superior a 2.0.
4	<i>bDeviceClass</i>	1byte	Classe	Código da classe.
5	<i>bDeviceSubClass</i>	1byte	Subclasse	Código da subclasse.
6	<i>bDeviceProtocol</i>	1byte	Protocolo	Código do protocolo.
7	<i>bMaxPacketSize0</i>	1byte	Quantidade	Tamanho máximo do pacote para outra velocidade. <i>Observação:</i> Valor especificado para o <i>endpoint</i> 0.
8	<i>bNumConfigurations</i>	1byte	Número	Número de configurações para outra velocidade.
9	<i>bReserved</i>	1byte	0x00	Reservado para uso futuro.

Se esse tipo de solicitação for enviado para um dispositivo que opera apenas no modo *Full-Speed*, este deve responder com um erro de requisição.



O host não deve solicitar o acesso ao descritor *other_speed_configuration*, se a solicitação do descritor *device_qualifier* reportar um erro.

18.10.4 Endpoint Descriptor

O *endpoint descriptor* fornece informações necessárias para a comunicação com um *endpoint*. Com essa informação o *host* determina a largura de banda requerida por cada *endpoint*. Esse descritor não pode ser acessado diretamente, sendo retornado como parte da informação retornada por uma solicitação do descritor de configuração.

As interfaces que usam mais de um *endpoint* têm um descritor de *endpoint* para cada um deles.

Tabela 18.15: Formato do *endpoint descriptor*.

Offset	Campo	Tamanho	Valor	Descrição
0	bLength	1byte	Tamanho	Tamanho do descritor.
1	bDescriptorType	1byte	0x05	Tipo do descritor.
2	bEndpointAddress	1byte	Endpoint	Endereço do <i>endpoint</i> do dispositivo USB.
3	bmAttributes	1byte	Bitmap	Atributos do <i>endpoint</i> .
4	wMaxPacketSize	2bytes	Quantidade	Tamanho máximo do pacote para esse <i>endpoint</i> .
6	blnterval	1byte	Número	Intervalo de chamada do <i>endpoint</i> para transferência de dados. Expressado em <i>frames</i> (1ms) ou <i>microframes</i> (125us), de acordo com a velocidade de operação.

O campo *bEndpointAddress* especifica o endereço do *endpoint* do dispositivo USB e possui o formato apresentado a seguir:

Tabela 18.16: Formato do campo *bEndpointAddress*.

Bit 7	Bit 6 a Bit 4	Bit 3 a Bit 0
Direção	Reservado	Número do endpoint
0 - OUT <i>endpoint</i> .		
1 - IN <i>endpoint</i> .	000	0 a 255

Observação: Campo ignorado pelo *endpoint* de controle

O campo *bmAttributes* especifica os atributos do *endpoint* quando ele é configurado usando o *bConfigurationValue*, o qual corresponde a um dos campos do descritor de configuração. Veja o seu formato em seguida.

Tabela 18.17: Formato do campo *bmAttributes*.

Bit 7 a Bit 6	Bit 5 a Bit 4	Bit 3 a Bit 2	Bit 1 a Bit 0
Reservado	Tipo usado	Tipo de sincronização	Tipo de transferência
00	00 = Data <i>endpoint</i> 01 = Feedback <i>endpoint</i> 10 = Implicit feedback Data <i>endpoint</i> 11 = Reservado Observação: Se não for um <i>isochronous endpoint</i> , esse campo deve ser 00	00 = Sem sincronização 01 = Assíncrono 10 = Adaptativo 11 = Síncrono Observação: Se não for um <i>isochronous endpoint</i> , esse campo deve ser 00.	00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt

O campo *wMaxPacketSize* especifica o tamanho máximo do pacote que o *endpoint* pode transmitir/receber, quando essa configuração é selecionada. Veja a seguir o seu formato.

Tabela 18.18: Formato do campo *wMaxPacketSize*.

<i>Bit 15 a Bit 13</i>	<i>Bit 12 a Bit 11</i>	<i>Bit 10 a Bit 0</i>
Reservado	Número de transações adicionais por <i>microframe</i>	Tamanho máximo do pacote
000	00 = 1 transação por <i>microframe</i> .	1 a 1024
	01 = 2 transação por <i>microframe</i> . (<i>Interrupt ou Isochronous</i>)	513 a 1024
	10 = 3 transação por <i>microframe</i> . (<i>Interrupt ou Isochronous</i>)	683 a 1024
	11 = Reservado.	Reservado

Se o dispositivo suportar o modo *High-Speed interrupt* ou *isochronous endpoint*, implica que ele necessita de uma largura de banda alta devido às elevadas taxas de dados, podendo chegar a 3072bytes (*Bit 12 e Bit 11 = 10* e *Bit 10 a Bit 0 = 1024*) de dados por *microframe*. Essa taxa é alcançada definindo várias transações *High-Speed* em um único *microframe* através dos *bits 11 e 12* do campo *wMaxPacketSize*, cuja combinação com os *bits 10 a 0* determina o número máximo de dados para cada transação.

Endpoint High-Speed isochronous e *interrupt* usam os *bits 11 e 12* para especificar múltiplas transações para cada *microframe* especificado pelo campo *bInterval*.

O campo *bInterval* especifica o intervalo de chamada do *endpoint* para transferência de dados e sua representação varia de acordo com o tipo de transferência e a velocidade do dispositivo. Observe a seguir.

- **Endpoint Full-Speed/High-Speed Isochronous:** o valor desse campo deve estar entre 1 e 16, e o período é representado por $2^{bInterval} - 1$.
- **Endpoint Full-Speed e Low-Speed Interrupt:** o valor desse campo deve estar entre 1 e 255, cujo período é representado pelo próprio valor do campo *bInterval*.
- **Endpoint High-Speed Interrupt:** o valor desse campo deve estar entre 1 e 16, e o período é representado por $2^{bInterval} - 1$.
- **Endpoint High-Speed Bulk ou OUT Control:** o valor desse campo deve estar entre 1 e 255, e especifica a taxa máxima de resposta NAK do *endpoint*. '0' indica que o dispositivo nunca responde com NAK e qualquer valor dentro do range válido indica uma resposta NAK a cada *bInterval* *microframes*.



O *endpoint 0* não possui um descritor, pois todos os dispositivos devem suportá-lo.

18.10.5 Interface Descriptor

O *interface descriptor* fornece informações sobre uma interface dentro de uma configuração. Uma interface nada mais é que um conjunto de *endpoints* utilizados por um recurso ou função do dispositivo, logo esse descritor fornece informações sobre os *endpoints* suportados pela interface.

Da mesma forma que o descritor de *endpoint*, o descritor de interface também não pode ser acessado diretamente, sendo retornado como parte da informação retornada por uma solicitação do descritor de configuração.

Tabela 18.19: Formato do *interface descriptor*.

Offset	Campo	Tamanho	Valor	Descrição
0	bLength	1byte	Tamanho	Tamanho do descritor.
1	bDescriptorType	1byte	0x04	Tipo do descritor.
2	blInterfaceNumber	1byte	Número	Número da interface.
3	bAlternateSetting	1byte	Número	Valor usado para selecionar esse recurso alternate para a interface identificada no campo anterior.
4	bNumEndpoints	1byte	Quantidade	Quantidade de <i>endpoints</i> utilizados por essa interface. (não inclui o <i>endpoint</i> 0)
5	blInterfaceClass	1byte	Classe	Código da classe.
6	blInterfaceSubClass	1byte	Subclasse	Código da subclasse.
7	blInterfaceProtocol	1byte	Protocolo	Código do protocolo.
8	iInterface	1byte	Índice	Índice do descritor de <i>string</i> , descrevendo essa interface.

Os campos *bDeviceClass*, *bDeviceSubClass* e *bDeviceProtocol* são complementares. A seguir é descrito o que cada um representa.

- » **blInterfaceClass:** especifica a classe do dispositivo.
 - 0x00: reservado para futuras aplicações.
 - 0x01 a 0xFE: indica que a interface suporta diferentes especificações de classe.
 - 0xFF: indica que a classe de interface é especificada pelo fabricante.
- » **blInterfaceSubClass:** especifica a subclasse do dispositivo.
 - 0x00: esse campo é setado em 0x00, se o campo *blInterfaceClass* for 0x00
 - 0x01 a 0xFE: identifica a subclasse do dispositivo de acordo com a sua classe (*blInterfaceClass*).
 - 0xFF: o valor da subclasse não é setado em 0xFF.
- » **blInterfaceProtocol:** especifica o protocolo usado pela interface com base na classe do dispositivo.
 - 0x00: indica que a interface não usa um protocolo especificado pela classe.
 - 0x01 a 0xFE: indica que a interface usa um protocolo especificado pela classe.
 - 0xFF: indica que a interface usa um protocolo especificado pelo fabricante.



Todos os códigos dos campos *blInterfaceClass*, *blInterfaceSubClass* e *blInterfaceProtocol* são definidos pela USB-IF e podem ser observados com maiores detalhes no site www.usb.org.

18.10.6 Other_Speed_Configuration Descriptor

O *other_speed_configuration descriptor* possui estrutura idêntica a *configuration descriptor*. A diferença é que ele fornece a configuração do dispositivo *High-Speed*, quando ele opera em uma velocidade que não está atualmente ativa.

Tabela 18.20: Formato do other_speed_configuration descriptor.

Offset	Campo	Tamanho	Valor	Descrição
0	bLength	1byte	Tamanho	Tamanho do descritor.
1	bDescriptorType	1byte	0x07	Tipo do descritor.
2	wTotalLength	2bytes	Quantidade	Quantidade total de dados retornados por essa configuração.
4	bNumInterfaces	1byte	Quantidade	Quantidade de interfaces suportadas por essa configuração de velocidade.
5	bConfigurationValue	1byte	Número	Valor usado como argumento para a solicitação SET_CONFIGURATION para selecionar essa configuração.
6	iConfiguration	1byte	Índice	Índice do descritor de string, descrevendo sua configuração.
7	bmAttributes	1byte	Bitmap	Características de configuração.
8	bMaxPower	1byte	mA	Máximo consumo de corrente pelo dispositivo Cada unidade equivale a 2mA.



O host não deve solicitar o acesso ao descritor *other_speed_configuration*, se a solicitação do descritor *device_qualifier* reportar um erro.

18.10.7 String Descriptor

O *string descriptor* fornece informações do fabricante, produto, número de serial, configuração e interface, em forma de texto.

A inclusão desse descritor é opcional, porém os campos pertencentes a ele são obrigatórios. Se o dispositivo não suportar descritores de *string*, os campos devem ser postos em zero, indicando a não disponibilidade desse descritor.

As *strings* que compõem o descritor podem suportar diversos idiomas e estão codificadas em UNICODE, em que a representação dos caracteres é de 16bits. O idioma é especificado pelo host ao enviar a solicitação de *string descriptor* e corresponde ao *Language ID (LANGID)* definido pelo USB-IF. Por exemplo, um LANGID = 0x0409 especifica o idioma inglês (Estados Unidos), LANGID = 0x040C especifica o idioma francês (padrão) e LANGID = 0x0416 representa o português (Brasil). O código correspondente a outros idiomas pode ser verificado no arquivo **USB_LANGIDs.pdf**, que pode ser baixado do site oficial www.usb.org.

Cada *string* é referenciada por um índice. Todos os dispositivos reservam a *string* 0 para informar os códigos (16bits) dos idiomas (LANGID) suportados pelo dispositivo em forma de uma matriz de códigos LANGID não finalizada com NULL. A Tabela 18.21 mostra os campos desse descritor.

Tabela 18.21: Formato do *string descriptor* para *string* 0.

Offset	Campo	Tamanho	Valor	Descrição
0	bLength	1byte	Tamanho	Tamanho do descritor.
1	bDescriptorType	1byte	0x03	Tipo do descritor.
2	wLANGID[0]	2bytes	Número	Código LANGID y.
...	Número	...
N	wLANGID[y]	2bytes	Número	Código LANGID y.

Se a *string* for 1 ou superior, o campo *bString* vai conter uma *string* codificada em UNICODE.

Tabela 18.22: Formato do *string descriptor* para *string* 1 ou superior

Offset	Campo	Tamanho	Valor	Descrição
0	<i>bLength</i>	1byte	Tamanho	Tamanho do descritor.
1	<i>bDescriptorType</i>	1byte	0x03	Tipo do descritor.
2	<i>bString</i>	N	Número	String codificada em UNICODE.

18.11 Classes

Em uma comunicação USB, dispositivos com atributos em comum são agrupados dentro de uma classe padronizada. Muitos sistemas operacionais (por exemplo, o Windows) já vêm equipados com *drivers* para as classes mais comuns, o que permite que dispositivos USB que pertencem a uma classe padronizada, cujo modo de operação é padrão, possam utilizar esses *drivers* para se comunicarem com o computador. Temos, por exemplo, o mouse e o teclado, ambos pertencentes à classe HID (*Human Interface Device*). Logo, se um determinado *firmware* for desenvolvido seguindo as especificações de um *driver* já disponível, o desenvolvedor não terá de se preocupar em criá-lo, apenas fazer algumas modificações.

A informação de uma classe é representada por 3bytes: classe, subclasse e protocolo, um byte por cada, podendo ser inserida no *Device Descriptor* ou *Interface Descriptors*. Veja a seguir alguns códigos relacionados às classes mais comuns.

Tabela 18.23: Código de algumas classes de uso genérico

Classe	Descritor	Descrição
0x01	Interface	Áudio
0x02	Device e Interface	Comunicação e controle CDC
0x03	Interface	HID (<i>Human Interface Device</i>)
0x06	Interface	Imagem
0x07	Interface	Impressora
0x08	Interface	Dispositivo de massa (exemplo HD)
0x09	Device	HUB
0x0A	Interface	CDC-DATA
0x0B	Interface	Smart Card
0x0D	Interface	Vídeo
0xE0	Interface	Controlador de wireless

Com o auxílio da tabela recém-mencionada podemos informar a classe de um determinado dispositivo HID pela representação a seguir.

Tabela 18.24: Exemplo de representação de uma classe HID.

Classe	Subclasse	Protocolo	Descrição
0x03	XX	XX	Classe de dispositivo HID.

Se o dispositivo for um teclado, teremos a seguinte representação.

Tabela 18.25: Representação da classe de um teclado.

Classe	Subclasse	Protocolo	Descrição
0x03	0x01	0x01	Teclado.

Note que o campo *Subclasse* é 0x01. Na classe HID, significa que o dispositivo suporta um protocolo predefinido, e o valor 0x01 no campo *Protocolo* especifica que o dispositivo é um teclado padrão.



Todas as especificações de classes, subclasses e protocolos podem ser obtidas no site USB-IF (USB Implementers Forum).

18.12 Requisições Padrão da USB

As requisições padrão da USB são comandos enviados pelo *host* e normalmente utilizados para habilitar/desabilitar recursos, verificar e/ou selecionar uma configuração específica, ler/atualizar/adicionar descritores, verificar o status do dispositivo, atribuir endereço ao dispositivo, entre outros. Os tipos de requisições estão listados na Tabela 18.4.

18.12.1 Clear_Feature

O CLEAR_FEATURE é utilizado para limpar ou desabilitar um recurso específico de um dispositivo, interface ou *endpoint*.

Tabela 18.26: Formato da solicitação CLEAR_FEATURE.

bmRequestType	bRequest	wValue	wIndex								wLength
			Byte 1		Byte 0						
			7	6	5	4	3	2	1	0	
0x00	0x01	Recurso	0x00							0x00	0x0000
0x01	0x01	Recurso	0x00							0x00 - 0xFF	0x0000
0x02	0x01	Recurso	0x00	D	0	0	0			0x00 - 0xF	0x0000

- **bmRequestType:** código do receptor.
 - 0x00: dispositivo.
 - 0x01: interface.
 - 0x02: *endpoint*.
- **bRequest:** código 0x01 correspondente à solicitação CLEAR_FEATURE.
- **wValue:** código do recurso que será desativado.

Tabela 18.27: Recursos padrão.

wValue	Nome	Receptor
0	ENDPOINT_HALT	<i>Endpoint</i>
1	DEVICE_REMOTE_WAKEUP	Dispositivo
2	TEST_MODE	Dispositivo



O recurso TESTE_MODE não pode ser limpo pela solicitação CLEAR_FEATURE.

- **wIndex:** se o receptor for um dispositivo, então o valor desse campo deve ser 0x0000. Se for uma interface, ele vai informar o número dela, podendo variar de 0x0000 a 0x00FF. No último caso, temos o *endpoint*, que uma vez selecionado, os 4bits menos significativos especificam o número do *endpoint* (0-15) e o bit 7 define a direção do fluxo de dados, sendo '1' para IN e '0' para OUT.
- **wLength:** sempre assume o valor 0x0000, pois não há retorno de dados.

18.12.2 Get_Configuration

O GET_CONFIGURATION solicita a configuração atual do dispositivo. Se o valor retornado for 0x00 significa que o dispositivo não está configurado.

Tabela 18.28: Formato da solicitação GET_CONFIGURATION

bmRequestType	bRequest	wValue	wIndex	wLength
0x80	0x08	0x0000	0x0000	0x0001

- **bRequest:** código 0x08 correspondente à solicitação GET_CONFIGURATION.
- **wValue:** valor constante. (0x0000)
- **wIndex:** valor constante. (0x0000)
- **wLength:** sempre assume o valor 0x0001, pois há retorno de apenas 1byte.

18.12.3 Get_Descriptor

O GET_DESCRIPTOR solicita o descriptor especificado, caso ele exista.

Tabela 18.29: Formato da solicitação GET_DESCRIPTOR.

bmRequestType	bRequest	wValue		wIndex	wLength
		Byte 1	Byte 0		
0x80	0x06	Tipo	Índice	0x0000 ou Language ID	Comprimento do descriptor

- **bRequest:** código 0x06 correspondente à solicitação GET_DESCRIPTOR
- **wValue:** o Byte 1 especifica o tipo de descriptor, de acordo com a Tabela 18.10, e o Byte 0 indica o valor do descriptor.
- **wIndex:** para descritores de string, esse campo identifica o idioma da string, de acordo com o código LANGID. Em qualquer outro tipo de descriptor, esse campo deve ser 0x0000.
- **wLength:** especifica a quantidade de bytes que serão retornados.

18.12.4 Get_Interface

O GET_INTERFACE solicita a definição alternativa selecionada da interface escolhida.

Tabela 18.30: Formato da solicitação GET_INTERFACE.

bmRequestType	bRequest	wValue	wIndex	wLength
0x81	0x0A	0x0000	Interface	0x0001

- **bRequest:** código 0x0A correspondente à solicitação GET_INTERFACE.
- **wValue:** valor constante. (0x0000)
- **wIndex:** especifica o número da interface, podendo variar de 0x0000 a 0x00FF.
- **wLength:** sempre assume o valor 0x0001, pois há retorno de apenas 1byte.

18.12.5 Get_Status

O GET_STATUS solicita o status do receptor especificado.

Tabela 18.31: Formato da solicitação GET_STATUS.

bmRequestType	bRequest	wValue	wIndex								wLength	
			Byte 1	Byte 0								
7	6	5	4	3	2	1	0					
0x80	0x00	0x0000	0x00									0x0002
0x81	0x00	0x0000	0x00									0x0002
0x82	0x00	0x0000	0x00	D	0	0	0					0x0002

- **bmRequestType:** código do receptor.
 - 0x80: dispositivo.
 - 0x81: interface.
 - 0x82: *endpoint*.
- **bRequest:** código 0x00 correspondente à solicitação GET_STATUS.
- **wValue:** valor constante. (0x0000)
- **wIndex:** se o receptor for um dispositivo, o valor desse campo deve ser 0x0000. Se for uma interface, ele vai informar o número dela, podendo variar de 0x0000 a 0x00FF. No último caso, temos o *endpoint*, que uma vez selecionado, os 4bits menos significativos especificam o número do *endpoint* (0-15) e o bit 7 define a direção do fluxo de dados, sendo '1' para IN e '0' para OUT.
- **wLength:** sempre assume o valor 0x0002, pois há retorno de apenas 2bytes. Veja a Tabela 18.32.

Tabela 18.32: Formato do pacote retornado.

Receptor	Byte 1		Byte 0	
	Bit 15 ao Bit 8	Bit 7 ao Bit 2	Bit 1	Bit 0
Dispositivo	00000000b	000000b	Remote Wakeup	Self Powered
Interface	00000000b	000000b	0	0
Endpoint	00000000b	000000b	0	Halt

Quando receptor é um dispositivo, temos os campos *Remote Wakeup* e *Self Powered*, que correspondem aos bits 1 e 0 do pacote retornado respectivamente.

Se o pacote retornado pelo dispositivo possuir o campo *Remote Wakeup* setado em '1', significa que esse recurso está habilitado; caso contrário, implica que ele está desabilitado. O mesmo vale para o campo *Self Powered*.

Quando o receptor é uma interface, o par de bytes possui o valor 0x0000.

Quando o receptor é um *endpoint*, o pacote retornado pelo dispositivo informa o status de apenas um recurso, conhecido como *Halt*. Se o campo *Halt* for '1', implica que o recurso está habilitado; caso contrário, indica que está desabilitado.

18.12.6 Set_Address

O SET_ADDRESS atribui um endereço para o dispositivo USB que será usado em todo o acesso.

Tabela 18.33: Formato da solicitação SET_ADDRESS.

bmRequestType	bRequest	wValue	wIndex	wLength
0x00	0x05	Endereço do dispositivo	0x0000	0x0000

- **bRequest:** código 0x05 correspondente à solicitação SET_ADDRESS.
- **wValue:** especifica o endereço do dispositivo. Lembre-se de que esse valor não deve ser superior a 127.
- **wIndex:** valor constante. (0x0000)
- **wLength:** sempre assume o valor 0x0000, pois não há transmissão de dados.

18.12.7 Set_Configuration

O SET_CONFIGURATION seleciona uma configuração do dispositivo.

Tabela 18.34: Formato da solicitação SET_CONFIGURATION.

bmRequestType	bRequest	wValue		wIndex	wLength
		Byte 1	Byte 0		
0x00	0x09	0x00	Valor de configuração	0x0000	0x0000

- **bRequest:** código 0x09 correspondente à solicitação SET_CONFIGURATION.
- **wValue:** especifica a configuração desejada, podendo variar de 0x00 a 0xFF. Se esse campo for 0x0000, o dispositivo é posto em modo de endereçamento.
- **wIndex:** valor constante. (0x0000)
- **wLength:** sempre assume o valor 0x0000, pois não há transmissão de dados.

18.12.8 Set_Descriptor

O SET_DESCRIPTOR é opcional e pode ser usado para atualizar ou adicionar descritores.

Tabela 18.35: Formato da solicitação SET_DESCRIPTOR.

bmRequestType	bRequest	wValue		wIndex	wLength
		Byte 1	Byte 0		
0x00	0x07	Tipo	Índice	0x00 ou Language ID	Comprimento do descritor.

- **bRequest:** código 0x07 correspondente à solicitação SET_DESCRIPTOR.
- **wValue:** o Byte 1 especifica o tipo de descritor, de acordo com a Tabela 18.10, e o Byte 0 indica o valor do descritor.
- **wIndex:** para descritores de *string*, esse campo identifica o idioma da string, de acordo com o código LANGID. Em qualquer outro tipo de descritor, esse campo deve ser 0x0000.
- **wLength:** especifica a quantidade de bytes que será transferida do host para o dispositivo.

18.12.9 Set_Feature

O SET_FEATURE seleciona ou habilita um recurso específico de um dispositivo, interface ou endpoint.

Tabela 18.36: Formato da solicitação SET FEATURE.

bmRequestType	bRequest	wValue	wIndex								wLength	
			Byte 1	Byte 0								
				7	6	5	4	3	2	1	0	
0x00	0x03	Recurso	Teste	0x00								0x0000
0x01	0x03	Recurso	0x00	0 - 0xFF								0x0000
0x02	0x03	Recurso	0x00	D	0	0	0	0	0	0	0 - 0xF	0x0000

- » **bmRequestType:** código do receptor.
 - 0x00: dispositivo.
 - 0x01: interface.
 - 0x02: *endpoint*.
- » **bRequest:** código 0x03 correspondente à solicitação SET_FEATURE.
- » **wValue:** código do recurso que será desativado, Tabela 18.27.
- » **wIndex:** se o receptor for um dispositivo, então o valor desse campo deve ser 0x0000. Se for uma interface, ele vai informar o número dela, podendo variar de 0x0000 a 0x00FF. No último caso, temos o *endpoint*, que uma vez selecionado, os 4bits menos significativos especificam o número do *endpoint* (0-15) e o bit 7 define a direção do fluxo de dados, sendo '1' para IN e '0' para OUT.
- » **wLength:** sempre assume o valor 0x0000, pois não há envio de dados.

18.12.10 Set_Interface

O SET_INTERFACE permite ao *host* selecionar uma configuração alternativa para a interface especificada.

Tabela 18.37: Formato da solicitação SET_INTERFACE.

bmRequestType	bRequest	wValue	wIndex	wLength
0x01	0x0B	Configuração alternativa	Interface	0x0000

- » **bRequest:** código 0x0B correspondente à solicitação SET_INTERFACE.
- » **wValue:** seleciona a configuração exclusiva da interface.
- » **wIndex:** especifica o número da interface, podendo variar de 0x0000 a 0x00FF.
- » **wLength:** sempre assume o valor 0x0000, pois não há envio de dados.

18.12.11 Synch_Frame

O SYNCH_FRAME seleciona um *endpoint* e solicita o retorno de um frame de sincronização. O frame de sincronização é suportado por alguns dispositivos High-Speed que usam a transferência *isochronous*, cujo tamanho dos pacotes pode variar de acordo com um padrão específico. Quando o dispositivo recebe essa solicitação, ele retorna o número do frame que precederá o início de uma nova sequência, ou seja, como o tamanho dos pacotes não é idêntico, é necessário que *host* saiba em que frame uma nova sequência será iniciada.

Tabela 18.38: Formato da solicitação SYNCH_FRAME.

bmRequestType	bRequest	wValue	wIndex								wLength	
			Byte 1	Byte 0								
				7	6	5	4	3	2	1	0	
0x82	0x0C	0x0000	0x00	D	0	0	0	0	0	0	0xF	0x0002

- » **bRequest:** código 0x0C correspondente à solicitação SYNCH_FRAME.
- » **wValue:** valor constante. (0x0000)
- » **wIndex:** os 4bits menos significativos especificam o número do *endpoint* (0-15) e o bit 7 define a direção do fluxo de dados, sendo '1' para IN e '0' para OUT.
- » **wLength:** sempre assume o valor 0x0002, pois há retorno de apenas 2bytes.

18.13 Processo de Enumeração do Dispositivo

Sempre que um dispositivo é conectado ao barramento USB, o *host* inicia um processo chamado de enumeração, que consiste em identificar e gerenciar o dispositivo. As etapas de um processo de enumeração podem ser vistas na Figura 18.32.

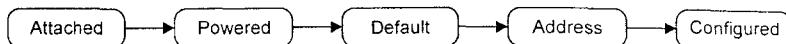


Figura 18.32: Etapas do processo de enumeração.

1. Quando um dispositivo é conectado ao barramento USB, o *hub* ou *root hub* informa ao *host* a ocorrência de um evento que é gerado pelo desbalanceamento das linhas D+ e D-, da porta *downstream* do *hub* ou *root hub*. Esse desbalanceamento é provocado pelo resistor *pull-up* do dispositivo. A partir desse ponto, o dispositivo encontra-se na etapa *Attached* (conectado) e a porta em que foi conectado é desativada.
2. O *host* solicita ao *hub* ou *root hub* verificar a natureza do evento.
3. Ao descobrir em que porta *downstream* do *hub* ou *root hub* o dispositivo foi conectado, o *host* aguarda pelo menos 100ms e então habilita essa porta, e envia um sinal de *reset* por um período de tempo tipicamente de 10ms.
4. Após o período de *reset*, o dispositivo pode drenar uma corrente de até 100mA fornecida pelo barramento. A partir de agora ele se encontra na etapa *Powered* (alimentado).
5. A próxima etapa é o *Default* (padrão), em que o *host* se comunica com o dispositivo com o endereço padrão (0x00) e o *endpoint* 0 (*Default Control Pipe*).
6. Através do endereço padrão e do *Default Control Pipe*, o *host* solicita a descrição do dispositivo por meio do *GET_DESCRIPTOR* com o objetivo de definir a quantidade máxima de dados que o *pipe* padrão pode manipular. Em seguida, atribui ao dispositivo um endereço único de 7bits (1 a 127) pela solicitação *SET_ADDRESS*, colocando o dispositivo na etapa *Address* (endereço).
7. O host solicita as configurações do dispositivo através do *GET_DESCRIPTOR*. Com base nessas informações e em como o dispositivo será usado, o *host* atribui um valor de configuração ao dispositivo por meio do *SET_CONFIGURATION*. Agora, o dispositivo se encontra na etapa *Configured* (configurado) e pronto para usar. Desse ponto em diante, o dispositivo pode drenar a quantidade de corrente especificada em seu descritor para a configuração selecionada.

18.14 Características do Módulo USB do PIC18F4550

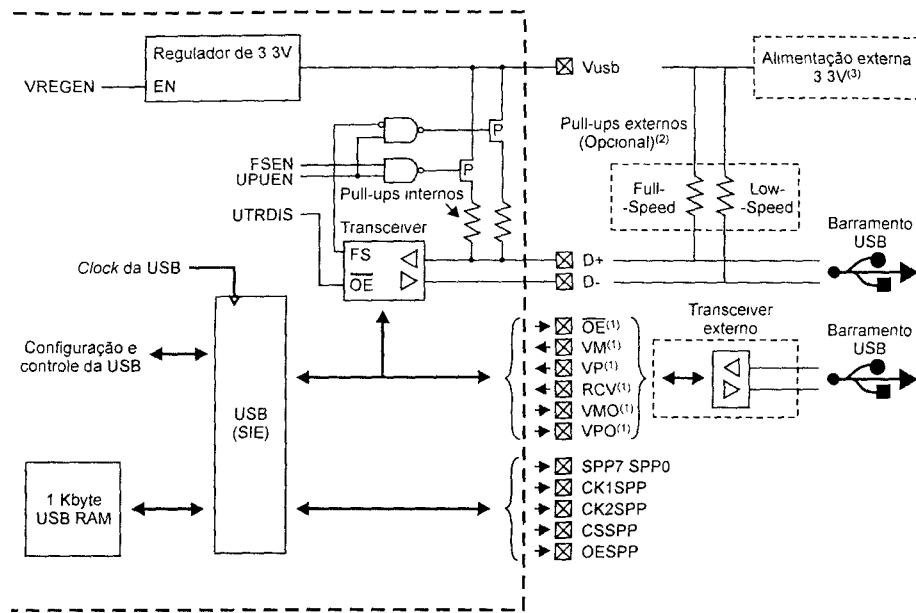
O microcontrolador PIC18F4550 possui um módulo USB 2.0 com uma *Serial Interface Engine* (SIE) compatível com os modos *Full-Speed* (12Mbps) e *Low-Speed* (1.5Mbps). A SIE do microcontrolador PIC18F4550 pode ser conectada ao barramento através de um *transceiver* interno (*UCFGbits.UTRDIS = 0*) ou externo (*UCFGbits.UTRDIS = 1*). Esse módulo também incorpora um regulador de tensão de 3,3V (V_{USB}), habilitado através da diretiva `#pragma config VREGEN = ON`, usado para alimentar o *transceiver* interno e dispõe de 1Kbyte de memória USB RAM acessada pela SIE, que compartilha o mesmo espaço de memória de dado acessado pela CPU (Banco 4 ao 7). No entanto, se uma seção estiver sendo utilizada pela SIE, não deve ser acessada pela CPU.

O USB suporta até 16 *endpoints* bidirecionais (0 a 15) e transferências do tipo *Control*, *Interrupt*, *Isochronous* e *Bulk*, além de dispor de um *Streaming Parallel Port* (SPP) capaz de transferir grande quantidade de dado de modo contínuo, tipicamente aplicado em transferências isócronas (*Isochronous transfers*). Outra característica interessante desse módulo é a disponibilidade de um conjunto de registros que descrevem os *buffers* (DB), localizados na USB RAM, que permitem ao usuário programá-los livremente.

O buffer do endpoint pode operar de três modos distintos: sem suporte ao *Ping-Pong Buffer*, com suporte ao *Ping-Pong Buffer* somente para OUT endpoint 0 ou com suporte ao *Ping-Pong Buffer* para todos os endpoints, além de ser capaz de armazenar dois pares completos de dados em cada direção, um par para transferência DATA 1 (IN e OUT) e um par para transferência DATA 0 (IN e OUT).

Esse módulo também possui o recurso *Suspend*, utilizado para reduzir o consumo do dispositivo USB alimentado pelo barramento, e o *Resume Signaling* que permite que o dispositivo realize um *wake-up* remotamente.

O regulador interno de 3.3V é usado para alimentar o transceiver interno e o resistor *pull-up* interno ou externo. Para garantir a estabilidade é necessário inserir um capacitor de 220nF ($\pm 20\%$) no pino Vusb do microcontrolador. A tensão de saída no Vusb deve ser usada apenas para alimentar o resistor *pull-up* externo, dada a corrente limitada.



Legenda

(1) Este sinal está disponível somente se o transceiver interno estiver desabilitado (UTRDIS = 1)

(2) Se resistores pull-ups forem inseridos externamente, não há necessidade de habilitar os pull-ups internos (UPUEN = 0 desabilitado).

(3) Não habilitar o regulador interno, quando estiver utilizando uma fonte externa de 3.3V.

Figura 18.33: Módulo USB interno do PIC18F4550.

Existem muitas fontes de interrupção no módulo USB. Por este motivo elas estão agrupadas em uma estrutura lógica específica do módulo, cujo estado de saída é uma simples interrupção de USB associada à lógica de interrupção do microcontrolador, em que o seu estado é definido pelo *Flag bit* de interrupção USBIF (*PIR<5>*).



Caso o leitor queira tomar nota dos registros desse módulo e do *Streaming Parallel Port* (SPP), os quais não estão comentados nesta obra, recomenda-se a leitura do *datasheet* do microcontrolador PIC18F4550.

18.14.1 Serial Interface Engine (SIE)

A SIE é responsável pela codificação/decodificação dos dados que trafegam no barramento. Por exemplo, codifica e converte os dados paralelos localizados no *buffer* (USB RAM) e envia-os serialmente para o barramento, como também decodifica e converte os dados recebidos e os transfere para o *buffer* (USB RAM), e podem ser acessados pela CPU do microcontrolador.

A SIE também é responsável pela geração e verificação do valor de CRC (Cyclic Redundancy Checks), o qual é utilizado como verificador de erros dos pacotes *token* e *data*, além de gerar o pacote de *handshake*. Ela também autoincrementa o endereço do *buffer*, supervisionado pela CPU.

18.14.2 Configuração do Oscilador para a SIE

Vimos que o módulo USB 2.0 interno do PIC18F4550 pode operar no modo *Full-Speed* ou *Low-Speed*. Porém, é necessário satisfazer as frequências impostas pela SIE a fim de garantir o seu correto funcionamento.

O modo *Full-Speed* necessita de um *clock* de 48MHz, obtido pelo oscilador primário de 48MHz ou pelo bloco 96MHz PLL interno do microcontrolador PIC18, cuja frequência de entrada é 4MHz e fornece uma saída de 96MHz que, dividida por 2, resulta 48MHz. A seleção da fonte de *clock* é definida pela diretiva **#pragma config CPUDIV**, sendo '2' para o bloco 96MHz PLL com o sinal de saída dividido por 2 e '1' para o oscilador primário sem *postscale*.

No modo *Low-Speed* é necessário alimentar a SIE com um *clock* de 6MHz, obtido por meio do *clock* primário de 24MHz, o qual é dividido por 4, resultando em um sinal de 6MHz.



O diagrama do *clock* pode ser visto no Capítulo 5.

18.15 Bibliotecas para a Comunicação USB

A Microchip Technology possui um pacote de bibliotecas chamado "*Microchip Application Libraries*", cujas bibliotecas podem ser utilizadas em uma grande variedade de aplicações, sendo uma delas a comunicação USB. Esse pacote pode ser baixado do site www.microchip.com e será necessário para o projeto proposto neste capítulo.

18.16 USB Hardware Abstraction Layer (HAL)

O pacote "*Microchip Application Libraries*" oferece um *firmware* para a USB capaz de funcionar em diferentes famílias de microcontroladores PIC®, com suporte a comunicação USB. Os nomes dos *Special Function Registers* (SFR) e de seus bits não são iguais, portanto é necessário especificar ao *firmware* a família do microcontrolador PIC® que está sendo utilizada no projeto, a fim de garantir a correta manipulação dos registros. Isso é feito pelo arquivo **usb_hal_picxx.h** que se encarrega de atribuir o nome apropriado para que as operações sejam executadas corretamente.

Como esta obra trata da família PIC18, o nome do arquivo é **usb_hal_pic18.h** e está localizado no diretório padrão:

"C:\Microchip Solutions\Microchip\Include\Usb\usb_hal_pic18.h"

Esse arquivo pode estar referenciado a partir do diretório em que foi instalado (diretório citado), ou ser copiado para dentro do projeto.

18.17 Configuração do Módulo USB

O arquivo `usb_config.h` agrupa as definições utilizadas na configuração do módulo USB para uma determinada aplicação. Ele fornece informações sobre o *Ping-Pong Buffer*, identificador do fabricante e produto, velocidade do módulo, tipo de operação, estado do resistor *pull-up* interno, *transceiver*, tamanho do buffer do endpoint 0, número de descritores de *string*, entre outras. Ele deve ser inserido em todos os projetos que utilizam as funções do pacote "Microchip Application Libraries", e pode ser facilmente gerado com o auxílio do software **Microchip USBConfig Utility**, localizado nesse mesmo pacote.

Após a instalação desse software no sistema operacional **Windows**, ele se encontra no seguinte caminho:

- **Inglês:** Start Menu → Programs → Microchip → MCHPFSUSB → Tools → USBConfig
- **Português:** Menu Iniciar → Programas → Microchip → MCHPFSUSB → Tools → USBConfig

Veja a seguir como configurar o módulo USB para ser reconhecido pelo *host* como uma porta serial COM virtual (CDC - *Serial Emulator*).

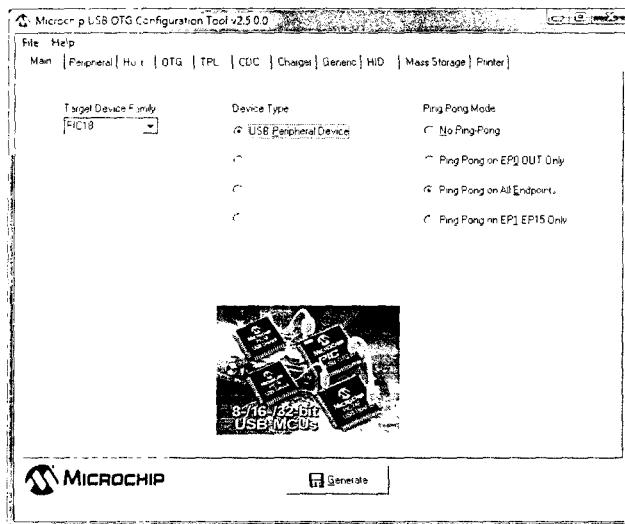


Figura 18.34: Tela principal do software **Microchip USBConfig Utility**.

A tela principal do software **Microchip USBConfig Utility**, apresentada na Figura 18.34, permite a especificação da família do microcontrolador PIC®, tipo de dispositivo USB e o modo de suporte ao *Ping-Pong Buffer*.

Neste caso, os campos devem estar selecionados do seguinte modo:

- **Target Device Family:** PIC18.
- **Device Type:** USB Peripheral Device.
- **Ping-Pong Mode:** Ping-Pong on All Endpoints.

Estes campos especificam que o microcontrolador pertence à família PIC18, vai operar como um periférico USB e o modo *Even/Odd Ping-Pong Buffer* (DATA0/DATA1) estará ativo para todos os *endpoints*.

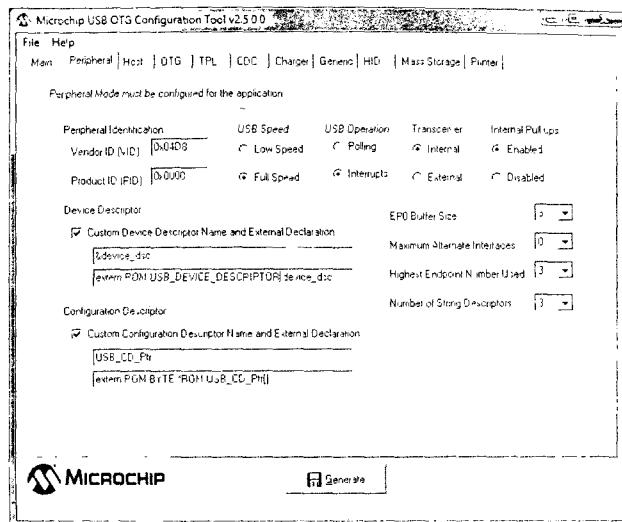


Figura 18.35: Campo de configuração do periférico.

Na tela "*Peripheral*" se encontram os campos necessários à configuração do periférico USB, que no caso devem ser configurados do modo apresentado a seguir.

- » **Peripheral Identification:** define um código relacionado à identificação do fabricante e do produto. (Exemplo: VID = 0x04D8 e PID = 0x0000)
- » **USB Speed:** define a velocidade do módulo. (*Full-Speed*)
- » **USB Operation:** operação da USB. (*Interrupts*)
- » **Transceiver:** tipo de *transceiver*. (*Internal*)
- » **Internal Pull-ups:** estado dos resistores *pull-ups* internos. (*Enabled*)
- » **Device Descriptor:** descritor de dispositivo.
- » **Configuration Descriptor:** descritor de configuração.
- » **EP0 Buffer Size:** tamanho do *buffer* do *endpoint* 0. (8)
- » **Maximum Alternate Interfaces:** quantidade máxima de interfaces alternativas. (0)
- » **Highest Endpoint Number Used:** número máximo do *endpoint*. (3)
- » **Number of String Descriptors:** quantidade de descritores de *String*. (3)



Todos os descritores devem ser declarados no arquivo `usb_descriptors.c`.

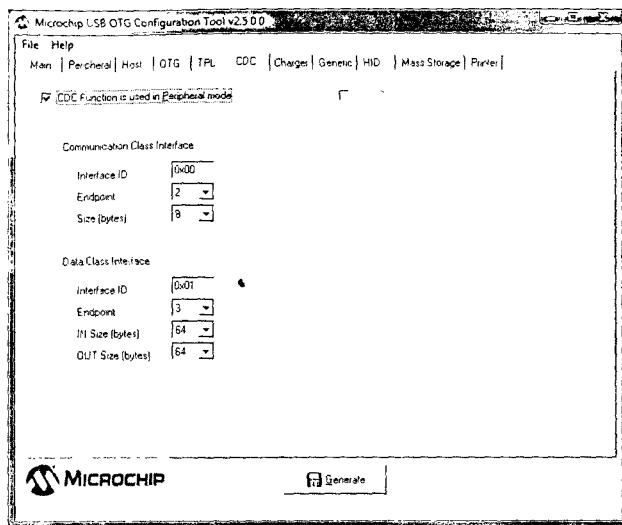


Figura 18.36: Campo de configuração da classe CDC.

A tela "CDC" compreende os campos de configuração das funções CDC.

- **Communication Class Interface:** configura a interface de comunicação.
 - **Interface ID:** identificação da interface. (0x00)
 - **Endpoint:** número do endpoint para a interface de comunicação. (2)
 - **Size:** tamanho do endpoint da interface. (8)
- **Data Class Interface:** configura a interface de dado.
 - **Interface ID:** identificação da interface. (0x01)
 - **Endpoint:** número do endpoint para a interface de dado. (3)
 - **IN Size:** tamanho máximo do pacote em uma transação IN. (64)
 - **OUT Size:** tamanho máximo do pacote em uma transação OUT. (64)

Após finalizar as três etapas de configuração, basta clicar no botão **Generate** e definir a pasta em que os arquivos **usb_config.h** e **usb_config.c** serão armazenados.

18.18 Arquivo de Descritores

Os descritores do periférico USB devem ser declarados com uma estrutura predefinida para que as funções pertencentes às bibliotecas USB possam reconhecê-los. As estruturas, constantes e macros que são usados para dar suporte ao protocolo estão localizados no arquivo **usb_ch9.h**.

"C:\Microchip Solutions\Microchip\Include\Usb\usb_ch9.h"

Veja, por exemplo, o arquivo **usb_descriptors.c** que declara os descritores para uma aplicação USB CDC.

"C:\Microchip Solutions\USB Device - CDC - Serial Emulator\CDC - Serial Emulator\usb_descriptor.c"



O arquivo **usb_descriptors.c** será utilizado no projeto proposto deste capítulo.

18.19 Funções de Controle da USB

O arquivo **usb_device.c** contém funções, macros, definições, variáveis, tipo de dados, entre outros, que são necessários para usar com a pilha do dispositivo. Ele está localizado no diretório padrão:

"C:\Microchip Solutions\Microchip\Usb\usb_device.c"

Vejamos na sequência a lista das funções declaradas nesse arquivo, bem como as suas descrições.

18.19.1 USBDeviceInit

Essa função inicia a pilha do dispositivo para o estado padrão, ou seja, todas as variáveis internas, os registros e os *Flag bits* de interrupção serão reiniciados.

Sintaxe

USBDeviceInit ()

18.19.2 USBDeviceTasks

Essa função é responsável pelo gerenciamento das tarefas do dispositivo USB. O modo de trabalhar depende diretamente do método utilizado. Existem dois métodos, o *Polling* e o *Interrupt*.

No método *Polling*, essa função deve ser chamada periodicamente em um tempo, de preferência, menor ou igual a 100us. Isso se deve ao fato de que o dispositivo USB deve verificar o status do barramento em um período de tempo inferior ao esperado para que o host encaminhe um pacote de SETUP ou OUT.

No método *Interrupt*, não há necessidade de chamar essa função periodicamente, pois qualquer atividade no barramento USB gera uma interrupção, e será verificada a fonte da interrupção para tratá-la. Lembramos que nesse método é necessário o uso das funções **USBDeviceAttach** e **USBDeviceDetach** para notificar a pilha quando o dispositivo está conectado ou desconectado do barramento USB.



Na maioria dos casos, essa função não leva mais que 50 ciclos de instrução para ser processada.

Sintaxe

USBDeviceTasks ()

Exemplo

//Método Polling - Verifica o status do barramento e serviços de interrupção periodicamente.

```
void main (void)
{
    ...
    USBDeviceInit(); // Inicia o módulo USB.

    while(1)
    {
        //No método polling, esta função deve ser chamada periodicamente, de preferência a cada 100us
        USBDeviceTasks();
        ...
    }
}
```

```

/*
Método Interrupt - Não há necessidade de verificar o status do barramento periodicamente, pois qualquer atividade no
barramento vai gerar uma interrupção, e dentro desta interrupção estará a função USBDeviceTasks( ).

void ISR_alta_prioridade(void); //Protótipo da função de interrupção.
...
#pragma code int_alta=0x08 //Vetor de interrupção de alta prioridade, ou padrão.
void int_alta (void)
{
    _asm GOTO ISR_alta_prioridade _endasm //Desvia o programa para a função ISR_alta_prioridade
}
#pragma code

#pragma interrupt ISR_alta_prioridade
void ISR_alta_prioridade(void)
{
    // Verifica a fonte da interrupção, trata e limpa o Flab bit.
    // Esta função é responsável pelo gerenciamento das tarefas do dispositivo USB.
    // Transações de SETUP ou OUT.
    USBDeviceTasks();
}

void main (void)
{
    ...
    USBDeviceInit( );// Inicia o módulo USB.

    // Informa à pilha (stack) que o dispositivo está conectado no barramento USB.
    // Esta função deve ser chamada quando a opção USB_INTERRUPT está selecionada.
    USBDeviceAttach( );

    PIR2bits.USBIF = 0; // Limpa o Flag bit da interrupção USB.
    IPR2bits.USBIPI = 1; // Interrupção USB - Alta prioridade.

    //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo
    RCONbits.IPEN = 1;
    INTCONbits.GIEH = 1;//Habilita todas as interrupções de alta prioridade.
    INTCONbits.GIEL = 1;//Habilita todas as interrupções de baixa prioridade.

    while(1)
    {
        ...
    }
}

```

18.19.3 USBEnableEndpoint

Ativa o *endpoint* com a configuração especificada.

Sintaxe

USBEnableEndpoint (*endpoint*, *configuração*)

Sendo:

- **endpoint:** informa o *endpoint* que será configurado e habilitado. Veja as constantes válidas no arquivo **USB_device.h**, localizado em "C:\Microchip Solutions\Microchip\Include\Usb".
- **configuração:** configuração opcional para o *endpoint*. As constantes válidas estão listadas na Tabela 18.39 e devem ser separadas por '|'.

Tabela 18.39 Constantes para o argumento configuração.

Constante	Descrição
USB_HANDSHAKE_ENABLED	Habilita o pacote de <i>handshake</i> (ACK e NAK).
USB_HANDSHAKE_DISABLED	Desabilita o pacote de <i>handshake</i> (ACK e NAK).
USB_OUT_ENABLED	Habilita a direção OUT.
USB_OUT_DISABLED	Desabilita a direção OUT.
USB_IN_ENABLED	Habilita a direção IN.
USB_IN_DISABLED	Desabilita a direção IN.
USB_ALLOW_SETUP	Habilita transferências de controle.
USB_DISALLOW_SETUP	Desabilita transferências de controle.
USB_STALL_ENDPOINT	Coloca o <i>endpoint</i> no modo STALL

Exemplo

```
USBEnableEndpoint( MSD_DATA_IN_EP, USB_IN_ENABLED | USB_OUT_ENABLED | USB_HANDSHAKE_ENABLED |  
USB_DISALLOW_SETUP );
```

18.19.4 USBStallEndpoint

Coloca o *endpoint* especificado no modo STALL, sendo a constante OUT_FROM_HOST para um OUT *endpoint* e IN_TO_HOST para um IN *endpoint*.

Sintaxe

```
USBStallEndpoint( endpoint, direção )
```

Sendo:

- **endpoint:** informa o *endpoint* que será afetado. Veja as constantes válidas no arquivo **USB_device.h**, localizado em "C:\Microchip Solutions\Microchip\Include\Usb".
- **direção:** direção da transferência. (OUT_FROM_HOST ou IN_TO_HOST)

Exemplo

```
USBStallEndpoint( EP0_OUT_EVEN, IN_TO_HOST );
```

18.19.5 USBTransferOnePacket

Transfere um pacote e retorna um ponteiro para o BDT (*Buffer Descriptor Table*) da transferência. Através desse ponteiro é possível ler o tamanho da última transferência, status e várias outras informações.

Sintaxe

```
BDT = USBTransferOnePacket( endpoint, direção, buffer, tamanho )
```

Sendo:

- **endpoint**: informa o endpoint que será usado. Veja as constantes válidas no arquivo **USB_device.h**, localizado em "C:\Microchip Solutions\Microchip\Include\Usb".
- **direção**: direção da transferência. (OUT_FROM_HOST ou IN_TO_HOST)
- **buffer**: ponteiro para os dados.
- **tamanho**: quantidade de dados.
- **BDT**: ponteiro para o BDT em uma transação IN_TO_HOST.

18.19.6 USBDeviceDetach

Informa à pilha (stack) que o dispositivo está desconectado do barramento USB.



Essa função deve ser usada pelo usuário quando a opção de interrupção está habilitada.

Sintaxe

USBDeviceDetach ()

Exemplo

```
// Informa à pilha (stack) que o dispositivo está desconectado no barramento USB
// Esta função deve ser chamada quando a opção USB_INTERRUPT está selecionada
USBDeviceDetach();
```

18.19.7 USBDeviceAttach

Informa à pilha (stack) que o dispositivo está conectado no barramento USB. Essa função deve ser usada pelo usuário quando a opção de interrupção está habilitada.



Essa função deve ser usada pelo usuário quando a opção de interrupção está habilitada.

Sintaxe

USBDeviceAttach ()

Exemplo

```
USBDeviceInit(); // Inicia o módulo USB.
// Informa à pilha (stack) que o dispositivo está conectado no barramento USB.
// Esta função deve ser chamada quando a opção USB_INTERRUPT está selecionada.
USBDeviceAttach();
```

18.20 Biblioteca USB CDC

A Microchip Technology dispõe de muitas bibliotecas voltadas à comunicação USB, porém neste livro somente a biblioteca **usb_function_cdc.c** é apresentada, pois ela dispõe de várias funções de suporte à classe CDC que serão utilizadas no projeto proposto neste capítulo, que consiste em simular uma porta COM virtual em um sistema operacional Windows 2000-XP.

Antes de comentarmos as funções de suporte para esse tipo de classe, veremos alguns passos necessários para facilitar a interface.

O primeiro passo é inserir os arquivos **usb_config.h**, **usb_descriptors.c**, **HardwareProfile.h**, **Compiler.h**, **GenericTypeDefs.h** e a pasta "USB" na pasta do projeto. Segue a descrição de cada arquivo ou pasta:

- » **usb_config.h**: esse arquivo contém as configurações do dispositivo USB e pode ser gerado pelo programa **Microchip USBConfig Utility**.
- » **usb_descriptors.c**: esse arquivo contém todos os descritores do dispositivo pertencente à classe CDC. Ele deve ser criado pelo usuário ou pode ser usado o modelo oferecido pela Microchip Technology.
"C:\Microchip Solutions\USB Device - CDC - Serial Emulator\CDC - Serial Emulator\usb_descriptor.c"
- » **HardwareProfile.h**: como o *firmware* disponibilizado pela Microchip foi desenvolvido para os seus KITS, para que ele funcione no projeto proposto no final deste capítulo é necessário adicionar o arquivo **HardwareProfile.h** à pasta do projeto. Esse arquivo tem a finalidade de informar a pilha sobre o hardware utilizado. Veja o código a seguir.

```
// Para que as funções operem de modo correto é necessário adicionar dois conjuntos de definições:
/*
O primeiro conjunto de definições está relacionado ao self_power. Este identificador é utilizado pelas funções como indicador da fonte de alimentação do dispositivo. Se o valor for '0', significa que ele está sendo alimentado pelo barramento; caso contrário, se for '1', indica que o dispositivo está sendo alimentado por uma fonte de tensão externa. O valor do self_power pode ser definido por um circuito externo que verifica a fonte de tensão e fornece o sinal para a entrada de um pino, ou pode ser simplesmente forçado a assumir um valor.
*/
// A definição seguinte deve ser comentada se nenhum pino estiver sendo usado para sinalizar a fonte de
// alimentação
#define USE_SELF_POWER_SENSE_IO

#if defined (USE_SELF_POWER_SENSE_IO)
    #define tris_self_power TRISAbits.TRISA2// Pino de entrada utilizado como indicador da fonte de alimentação.
                                            //Observação: Pode ser qualquer pino que possa ser configurado como entrada digital.
    #define self_power PORTAbits.RA2// Pino de entrada utilizado como indicador da fonte de alimentação.
                                            //Observação: Pode ser qualquer pino que possa ser configurado como entrada digital.
#else
    #define self_power 1
#endif

/*
O segundo conjunto de definições está relacionado ao USB_BUS_SENSE. Este identificador é utilizado para informar se o dispositivo está ou não conectado no barramento USB. Se USB_BUS_SENSE for '0', implica que o dispositivo está desconectado e se for '1', indica que o dispositivo está conectado no barramento. Da mesma forma que o self_power, o valor deste identificador pode ser obtido através de um circuito externo, ou pode-se definir um valor constante a ele, nos casos em que não há um pino destinado a esta função.
*/
// A definição seguinte deve ser comentada se nenhum pino estiver sendo usado para sinalizar se o dispositivo está
// conectado no barramento.
#define USE_USB_BUS_SENSE_IO

#if defined(USE_USB_BUS_SENSE_IO)
    #define tris_usb_bus_sense TRISBbits.TRISB5 // Pino de entrada utilizado como indicador de conexão.
                                                //Observação: Pode ser qualquer pino que possa ser configurado como entrada digital.
    #define USB_BUS_SENSE PORTBbits.RB5        // Pino de entrada utilizado como indicador de conexão.
                                                //Observação: Pode ser qualquer pino que possa ser configurado como entrada digital.
```

```
#else
#define USB_BUS_SENSE 1
#endif

// Indica que a plataforma usada não é um dos KITs da Microchip.
#define DEMO_BOARD USER_DEFINED_BOARD
```

- Compiler.h, GenericTypeDefs.h e a Pasta "USB": ambos os arquivos estão localizados no diretório "C:\Microchip Solutions\Microchip\Include" e são fundamentais para o funcionamento do programa, pois possuem as definições e os arquivos básicos para prover comunicação USB.

O último detalhe importante é que os arquivos de nome **usb_function_cdc.c** e **usb_device.c** devem ser adicionados ao projeto, pois o primeiro contém as funções de suporte à classe CDC e o segundo dispõe das funções básicas de tratamento das solicitações da USB. Veja a Figura 18.37.

- usb_device.c**: está localizado no diretório "C:\Microchip Solutions\Microchip\Usb".
- usb_function_cdc**: encontra-se disponível no diretório "C:\Microchip Solutions\Microchip\Usb\CDC Device Driver".

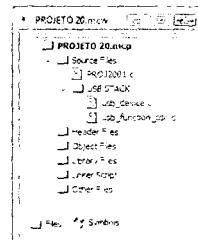


Figura 18.37: Modelo dos arquivos adicionados ao projeto.



O drive USB-CDC está localizado no diretório "C:\Microchip Solutions\USB Device - CDC - Serial Emulator\inf\win2k_winxp".

18.20.1 USBCheckCDCRequest

Essa função verifica se a solicitação enviada pelo *host* é suportada pela rotina. Caso seja, ela é tratada.

Sintaxe

USBCheckCDCRequest ()

18.20.2 CDCInitEP

Essa função inicia os *endpoints*, seta o *baud rate*, *bit parity*, *stop bit*, número de *bits* de dados e prepara o dispositivo para a primeira transferência a partir do *host*.

A configuração do *baud rate*, *bit parity*, *stop bit* e o número de *bits* de dados pode ser realizada modificando os campos das estruturas localizadas dentro dessa função. São elas:

- line_coding.dwDTERate.Val = 9600;** // *Baud rate*.
- line_coding.bCharFormat = 0x00;** // *Stop bit*.
- line_coding.bParityType = 0x00;** // *Bit Parity*.
- line_coding.bDataBits = 0x08;** // Quantidade de *bits* de dados.

Os valores possíveis para a estrutura de *Stop bit* são:

- 0x00:** 1 Stop bit.
- 0x01:** 1,5 Stop bit.
- 0x02:** 2 Stop bit.

Os valores possíveis para a estrutura de *bit parity* são:

- 0x00: nenhum.
- 0x01: Odd.
- 0x02: Even.
- 0x03: Mark.
- 0x04: Space.

Os valores possíveis para a estrutura de quantidade de *bits* de dados são:

- 0x05: 5bits.
- 0x06: 6bits.
- 0x07: 7bits.
- 0x08: 8bits.
- 0x10: 16bits.



Essa função deve chamada logo após o comando SET_CONFIGURATION.

Sintaxe
CDCInitEP()

18.20.3 getsUSBUSART

Recebe uma *string* de caracteres enviada pela USB CDC Bulk OUT endpoint e a copia para uma região da memória de dados definida pelo usuário. Essa função não fica presa à rotina caso não seja verificada a disponibilidade de dados no momento. e retorna um '0' se isso for verificado; caso contrário, retorna a quantidade de caracteres recebidos.

Sintaxe
quant_receb = getsUSBUSART (dados_recebidos, quantidade)

Sendo:

- **quant_receb**: quantidade de caracteres recebidos.
- **dados_recebidos**: ponteiro para o local da memória de dados onde os dados serão armazenados.
- **quantidade**: quantidade de caracteres que podem ser recebidos pela função. (0 - 255)

Exemplo

```
char quanti_dados_recebidos; //Variável utilizada para informar a quantidade de bytes recebidos.  
char buffer[16]; //Matriz de 16 elementos do tipo char.
```

```
quanti_dados_recebidos = getsUSBUSART(buffer,sizeof(buffer));
```

18.20.4 putUSBUSART

Envia uma matriz de dados para a USB. Ela também suporta o envio de um caractere NULL. Essa função deve ser usada em conjunto com a função **USBUSARTIsTxTrfReady** com o objetivo de evitar erros na transmissão, uma vez que esta última é responsável pela verificação do status do envio.

Sintaxe

```
putUSBUSART ( dados_envio, quantidade )
```

Sendo:

- ***dados_envio***: ponteiro para o local (memória de dados) onde os dados estão armazenados.
- ***quantidade***: quantidade de caracteres que serão transmitidos. (0 - 255)

Exemplo

```
char buffer [12] = {"Teste PIC18"};
```

```
if (USBUSARTIsTxTrfReady())//Verifica se o módulo pode transmitir.
```

```
    putUSBUSART (buffer,11); //Envia uma matriz de 11 elementos.
```

18.20.5 putsUSBUSART e putrsUSBUSART

Ambas as funções, **putsUSBUSART** e **putrsUSBUSART**, enviam uma matriz de dados para a USB incluindo um caractere NULL. Elas também suportam o envio de um caractere NULL. Essas funções devem ser usadas em conjunto com a função **USBUSARTIsTxTrfReady** com o objetivo de evitar erros na transmissão, uma vez que esta última é responsável pela verificação do status do envio.

A principal diferença entre elas é que a função **putsUSBUSART** envia dados localizados na memória de dados, enquanto a função **putrsUSBUSART** envia dados localizados na memória de programa.

Sintaxe

```
putsUSBUSART ( dados_ram_envio )
```

```
putrsUSBUSART ( dados_rom_envio )
```

Sendo:

- ***dados_ram_envio***: ponteiro para o local da memória de dados onde os dados estão armazenados.
- ***dados_rom_envio***: ponteiro para o local da memória de programa onde os dados estão armazenados.

Exemplo

```
char buffer [] = "Teste RAM";
```

```
if (USBUSARTIsTxTrfReady( )) //Verifica se o módulo pode transmitir.
```

```
    putsUSBUSART (buffer); //Envia uma string e finaliza com um caractere NULL.
```

```
if (USBUSARTIsTxTrfReady( ))//Verifica se o módulo pode transmitir.
```

```
    putrsUSBUSART ("Teste ROM"); //Envia uma string e finaliza com um caractere NULL.
```

18.20.6 CDCTxService

Essa função manipula as transações entre o dispositivo e o *host*. Ela deve ser chamada continuamente dentro da rotina principal, uma vez que o dispositivo já esteja configurado.

Sintaxe

```
CDCTxService ( )
```

Exemplo

```
...
```

```
While (1)
```

```
{
```

```
USBDeviceTasks(); //Esta função é responsável pelo gerenciamento das tarefas do dispositivo. (SETUP ou OUT)
```

```
if (USBUSARTIsTxTrfReady( )) //Verifica se o módulo pode transmitir.
```

```
putrsUSBUSART("PIC18F4550"); //Imprime uma string na USB - RS232
```

```
CDCTxService(); //Esta função manipula as transações entre o dispositivo e o host.  
Delay10KTCYx(250); //Gera um delay de 208.33 milissegundos. 250*10.000*0.0633333us = 208.33ms  
}
```

18.21 Projeto

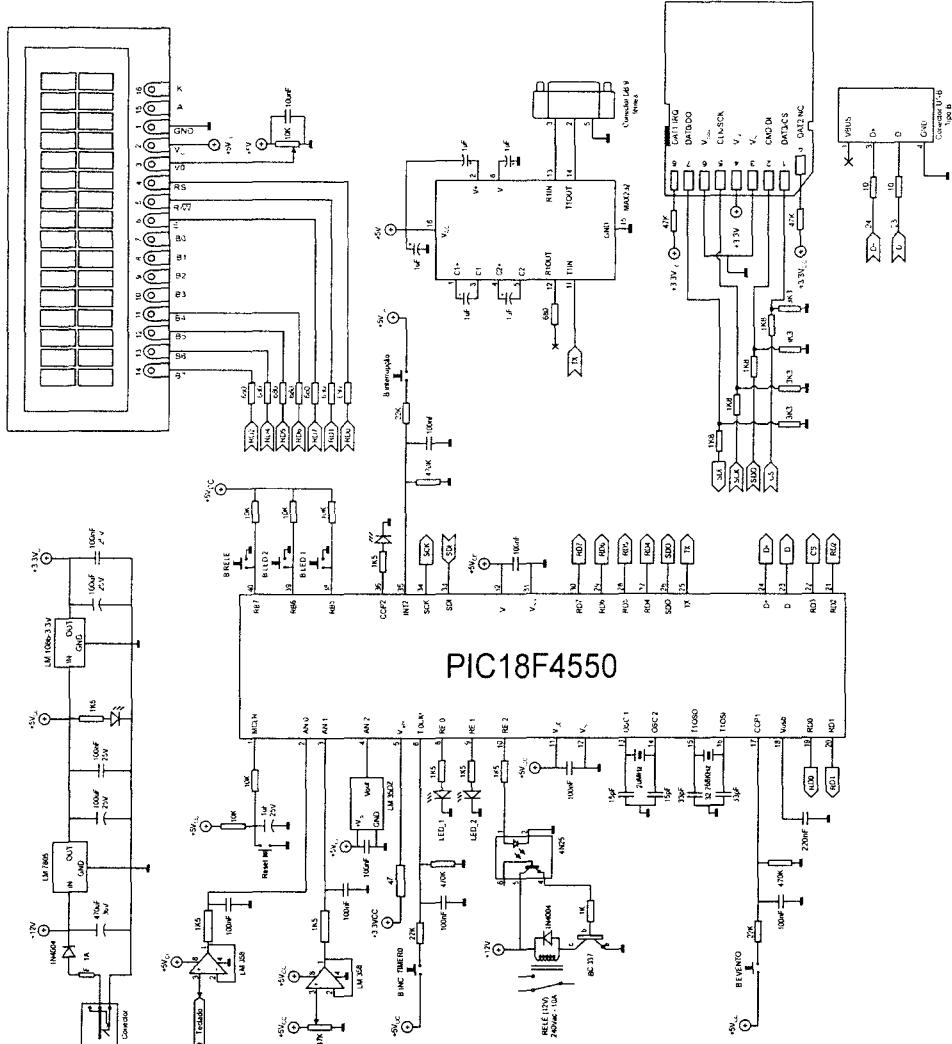


Figura 18.38: Circuito de comunicação USB, com alimentação dupla. (VBUS não está conectado com Vdd)

Código

```
*****Projeto Capítulo 18 - USB -> RS232*****
**
** Este programa cria uma porta COM (RS232) virtual, com taxa de transferência igual a 19200bps
** O microcontrolador envia o valor da temperatura ou a tensão de saída do potenciômetro,
** caso receba os caracteres 'P' - Potenciômetro e 'T' - Temperatura.
** O PIC18F4550 informa através do display LCD 2x16 qual tipo de informação está enviando pela USB
** Caso o comando enviado do host para o PIC não for válido (somente 'T' e 'P' são válidos),
** o display imprime a seguinte mensagem de erro: "CODIGO INCORRETO".
**
** Autor: Alberto Noboru Miyadaira
*****/
```

```
#include <p18f4550.h> //Arquivo de cabeçalho do PIC18F4550.
#include <adc.h>      //Adiciona a biblioteca de funções para o módulo conversor A/D.
#include <stdio.h>     //Adiciona a biblioteca padrão de entrada e saída.
#include <stdlib.h>    //Adiciona a biblioteca de funções miscelâneas.
#include <string.h>    //Adiciona a biblioteca de manipulação de string.
#include "C:\pic18\cabecalho\lcd_2x16.h" //Biblioteca contendo as funções do LCD.

// Inclusão de arquivos para a comunicação USB-CDC.
// Na pasta Source Files do projeto, devem ser adicionados dois arquivos, são eles:
// "C:\Microchip Solutions\Microchip\Usb\CDC Device Driver\usb_function_cdc.c"
// "C:\Microchip Solutions\Microchip\Usb\usb_device.c"
#include "C:\Microchip Solutions\Microchip\Include\GenericTypeDefs.h"
#include "usb_config.h"
#include "usb_descriptors.c"
#include "C:\Microchip Solutions\Microchip\Include\Usb\usb_device.h"
#include "C:\Microchip Solutions\Microchip\Include\Usb\usb.h"
#include "C:\Microchip Solutions\Microchip\Include\Usb\usb_function_cdc.h"
#include "HardwareProfile.h"

// Fcrtal = 20MHz
// Fosc = 48MHz - Fonte: 96MHz PLL/2
// Tciclo = 4/Fosc = 0,083us
#pragma config PLLDIV = 5
#pragma config CPUDIV = OSC1_PLL2
#pragma config USBDIV = 2
#pragma config FOSC = HSPLL_HS

#pragma config WDT = OFF      //Desabilita o Watchdog Timer (WDT).
#pragma config PWRT = ON       //Habilita o Power-up Timer (PWRT).
#pragma config BOR = ON        //Brown-out Reset (BOR) habilitado somente no hardware.
#pragma config BORV = 1         //Voltagem do BOR é 4,33V.
#pragma config PBADEN = OFF    //RB0,1,2,3 e 4 configurado como I/O digital.
#pragma config LVP = OFF        //Desabilita o Low Voltage Program.
#pragma config VREGEN = ON       //Habilita o regulador de tensão da USB.

//Variáveis globais
unsigned char buffer[14]; // Buffer de armazenamento dos dados enviados pela USB.
unsigned char c; //Armazena o caractere recebido pela USB.

void ISR_alta_prioridade(void); //Protótipo da função de interrupção.
```

```

float valor; // Armazena o valor devolvido pelo conversor A/D.
float valor_temperatura, //Armazena o valor da temperatura (LM35D)

//Converte o valor devolvido pelo conversor em um valor correspondente à tensão [mV].
float converte_tensao (float valor_conversor)
{
    return (valor_conversor*3300)/1023; //Neste caso, a tensão de entrada no pino Vref+ é 3.3V.
}

//Converte o valor de tensão em um valor correspondente à temperatura [C].
//Sensor de temperatura: LM35D - 10mV/C. Range de temperatura -55C a +150C.
float converte_temperatura (float valor_tens)
{
    return valor_tens/10;
}

//Filtro em software.
//Obtém 256 amostras e retorna a média
unsigned int filtro_canal()
{
    unsigned int cont_filtro;
    unsigned long valor_canal = 0;

    for( cont_filtro = 0 ; cont_filtro < 256 ; cont_filtro++ )
    {
        ConvertADC(); //Inicia a conversão.
        while (BusyADC()); //Aguarda o fim da conversão.
        valor_canal += ReadADC(); //Armazena o resultado da conversão
    }
    return (valor_canal >> 8); // Esta operação é igual a valor_canal/256.
}

#pragma code int_alta=0x08 //Vetor de interrupção de alta prioridade, ou padrão.
void int_alta (void)
{
    _asm GOTO ISR_alta_prioridade _endasm //Desvia o programa para a função ISR_alta_prioridade.
}
#pragma code

#pragma interrupt ISR_alta_prioridade
void ISR_alta_prioridade(void)
{
    char val;

    // Verifica a fonte da interrupção, trata e limpa o Flab bit.
    USBDeviceTasks(); //Esta função é responsável pelo gerenciamento das tarefas do dispositivo (SETUP ou OUT)

    // Verifica se o dispositivo está no estado configurado.
    // Se estiver configurado, executa as rotinas do usuário.
    // O identificador CONFIGURED_STATE é 0x20 e os demais estados possuem um valor inferior a este,
    // por isso, se o valor do USBDeviceState (estado atual) for menor, pula a interação.
    if(USBDeviceState < CONFIGURED_STATE) return 0;

    c = 0;
}

```

```

val = getsUSBUSART(&c, 1); //Lê o dado recebido pela USB(RS232).

if(val > 0)
{
    if( c == 'P' )
    {
        SetChanADC (ADC_CH1); //Carrega o canal analógico 1. (AN1) - Potenciômetro
        valor = filtro_canal (); //Chama a função para fazer a leitura do canal.

        //Converte o valor retornado pelo conversor A/D em um valor de tensão correspondente.
        valor = converte_tensao(valor);

        sprintf(buffer, "Pot: %04d mv", (int)valor); //Envia a string formatada para o buffer.

        if (USBUSARTIsTxTrfReady( )) //Verifica se o módulo pode transmitir.
            putsUSBUSART(buffer); //Imprime uma string na USB - RS232.

        lcd_posicao (2,1); // Desloca o ponteiro para a L=2 e C=1.
        imprime_string_lcd(" POTENCIOMETRO "); //Imprime uma string no display.
    }
    else
    {
        if( c == 'T' )
        {
            SetChanADC (ADC_CH2); //Carrega o canal analógico 2. (AN2) - Temperatura.
            valor = filtro_canal (); //Chama a função para fazer a leitura do canal.

            //Converte o valor retornado pelo conversor A/D em um valor de tensão correspondente
            valor_temperatura = converte_tensao(valor);
            valor_temperatura = converte_temperatura (valor_temperatura); //Converte a tensão em temperatura.

            sprintf(buffer, "Temp:      %03d.%02d      C", (char)valor_temperatura,(char)((valor_temperatura-
            (char)valor_temperatura)*100)); //Envia a string formatada para o buffer.

            if (USBUSARTIsTxTrfReady( )) //Verifica se o módulo pode transmitir.
                putsUSBUSART(buffer); //Imprime uma string na USB - RS232.

            lcd_posicao (2,1); // Desloca o ponteiro para a L=2 e C=1.
            imprime_string_lcd(" TEMPERATURA "); //Imprime uma string no display.
        }
        else
        {
            lcd_posicao (2,1); // Desloca o ponteiro para a L=2 e C=1.
            imprime_string_lcd(" CODIGO INCORRETO"); //Imprime uma string no display.
        }
    }
}

CDCTxService(); //Esta função manipula as transações entre o dispositivo e o host.

// Esta função é chamada pela pilha USB para notificar ao usuário a ocorrência de um evento USB.
// Ela é necessária quando a opção USB_INTERRUPT está selecionada, como é o caso deste exemplo.
// Veja o arquivo usb_config.h (gerado pelo software Microchip USBConfig Utility).
BOOL USER_USB_CALLBACK_EVENT_HANDLER(USB_EVENT event, void *pdata, WORD size)
// event - Tipo do evento.
// *pdata - Ponteiro para o dado do evento.

```

```

// size - Tamanho do dado do evento.
{
    switch(event)
    {
        // Este evento é disparado quando o dispositivo recebe uma solicitação
        // SET_CONFIGURATION enviada pelo host. (wValue != 0)
        case EVENT_CONFIGURED:
            // Inicia os endpoints para o uso do dispositivo de acordo com a configuração atual. (CDC)
            // Seta o baud rate, bit parity, stop bit e o número de bits de dados.
            // A configuração padrão está indicada a seguir.
            // Baud rate: 19.200 bps
            // Bit de parada: 1
            // Bit de paridade: sem
            // Número de bits do dado: 8 bits.
            // Para modificar esta configuração, basta abrir o arquivo usb_function_cdc.c e
            // modificar os campos da função CDCInitEP().
            CDCInitEP();
            break;

        // Este evento é disparado quando o dispositivo recebe um pacote de SETUP enviado pelo host,
        // e deve responder para completar a solicitação.
        case EVENT_EP0_REQUEST:
            USBCheckCDCRequest(); // Verifica e trata a solicitação.
            break;

        case EVENT_TRANSFER:
            Nop();
            break;

        default:
            break;
    }
    return 1;
}

void main(void) //Função principal.
{
    TRISA = 0b00011111; //RA0 a RA4 - entrada e RA5 a RA6 - saída.
    TRISB = 0b11100111; //RB0,RB1,RB2,RB5,RB6 e RB7 - entrada e RB3 a RB4 - saída
    TRISC = 0b10111111; //RC0 a RC5 e RC7 - entrada e RC6 - saída.
    TRISD = 0b00000000; //RD0 a RD7 - saída.
    TRISE = 0b00000000; //RE0 a RE2 - saída.

    // Define o pino como entrada, se o hardware possuir um pino com a finalidade de sinalizar a fonte de alimentação
    // do dispositivo. (HardwareProfile.h)
    #if defined(USE_USB_BUS_SENSE_IO)
        tris_usb_bus_sense = INPUT_PIN;
    #endif

    // Define o pino como entrada, se o hardware possuir um pino com a finalidade de sinalizar se o dispositivo está
    // conectado no barramento USB. (HardwareProfile.h)
    #if defined(USE_SELF_POWER_SENSE_IO)
        tris_self_power = INPUT_PIN;
    #endif
}

```

```

&ADC_RIGHT_JUST      //Resultado justificado para a direita.
&ADC_2_TAD,           //Configuração do tempo de aquisição automático. (2*Tad = 2.667us)
ADC_CH1,               //Seleciona o canal 1 (AN1)
&ADC_INT_OFF          //Interrupção desabilitada.
&ADC_VREFPLUS_EXT    //Vref+ = Vref+ (pin AN3)
&ADC_VREFMINUS_VSS,   //Vref- = Vss
ADC_6ANA);             //Habilita o canal AN0 a AN5.

```

```

lcd_inicia(0x28, 0x0F, 0x06); //Inicializa o display LCD alfanumérico com duas linhas de dados.
lcd_LD_cursor (0); //Desliga o cursor.

```

```

lcd_posicao (1,3); //Posiciona o cursor na L=1 e C=3;
imprime_string_lcd("USB -> RS232"); //Imprime uma string.

```

```

USBDeviceInit(); //Inicia o módulo USB.

```

```

// Informa à pilha (stack) que o dispositivo está conectado no barramento USB.
// Esta função deve ser chamada quando a opção USB_INTERRUPT está selecionada
USBDeviceAttach();

```

```

PIR2bits.USBIF = 0; // Limpa o Flag bit da interrupção USB.
PIR2bits.USBIPI = 1; // Interrupção USB - Alta prioridade.

```

```

RCONbits.IPEN = 1; //Habilita interrupção com nível de prioridade. Endereço 0x08 - alto e 0x18 - baixo
INTCONbits.GIEH = 1; //Habilita todas as interrupções de alta prioridade.
INTCONbits.GIEL = 1; //Habilita todas as interrupções de baixa prioridade

```

```

while (1); //Looping infinito.
}

```

Tabela ASCII

A tabela ASCII (*American Standard Code for Information Interchange*) define uma codificação para os caracteres a fim de permitir a troca de informações entre dispositivos. A tabela ASCII comum representa os dados em 7bits.

Hexa	Caractere	Hexa	Caractere	Hexa	Caractere	Hexa	Caractere
20	Espaço	38	8	50	P	68	h
21	!	39	9	51	Q	69	i
22	"	3A	:	52	R	6A	j
23	#	3B	;	53	S	6B	k
24	\$	3C	<	54	T	6C	l
25	%	3D	=	55	U	6D	m
26	&	3E	>	56	V	6E	n
27	'	3F	?	57	W	6F	o
28	(40	@	58	X	70	p
29)	41	A	59	Y	71	q
2A	*	42	B	5A	Z	72	r
2B	+	43	C	5B	[73	s
2C	,	44	D	5C	\	74	t
2D	-	45	E	5D]	75	u
2E	.	46	F	5E	^	76	v
2F	/	47	G	5F	_	77	w
30	0	48	H	60	`	78	x
31	1	49	I	61	a	79	y
32	2	4A	J	62	b	7A	z
33	3	4B	K	63	c	7B	{
34	4	4C	L	64	d	7C	
35	5	4D	M	65	e	7D	}
36	6	4E	N	66	f	7E	~
37	7	4F	O	67	g	7F	Delete

Existem caracteres que não são imprimíveis, conhecidos como caracteres de controle. A tabela seguinte lista os caracteres de controle e suas descrições, além das respectivas constantes de barra invertida, que podem ser utilizadas em funções `printf`.

Hexa	Caractere	Descrição	Comando utilizado nas funções printf
00	NUL	Nulo.	\0
01	SOH	Começo do cabeçalho.	-
02	STX	Começo do texto.	-
03	ETX	Fim do texto.	-
04	EOT	Fim da transmissão.	-
05	ENQ	Interroga identidade do terminal.	-
06	ACK	Confirmação.	-
07	BEL	Campainha.	\a
08	BS	Retrocesso. (Backspace)	\b
09	HT	Tabulação horizontal.	\t
0A	LF	Nova linha.	\n
0B	VT	Tabulação vertical.	\v
0C	FF	Nova página.	\f
0D	CR	Retorno do carro. (Tecla ENTER)	\r
0E	SO	Shift-out.	-
0F	SI	Shift-in.	-
10	DLE	Data link escape.	-
11	D1	Controle do dispositivo.	-
12	D2	Controle do dispositivo.	-
13	D3	Controle do dispositivo.	-
14	D4	Controle do dispositivo.	-
15	NAK	Não confirmação.	-
16	SYN	Synchronous idle.	-
17	ETB	Fim da transmissão do bloco.	-
18	CAN	Cancela.	-
19	EM	End-Of-Medium.	-
1A	SUB	Substitui.	-
1B	ESC	Escape. (Tecla ESC)	-
1C	FS	File Separator.	-
1D	GS	Group Separator.	-
1E	RS	Record Separator.	-
1F	US	Unit Separator.	-

Bibliografia

- AXELSON, J. **USB Complete. Everything You Need to Develop Custom USB Peripherals.** 3th Edition. Madison: Lakeview Research LLC, 2005.
- SCHILD'T, H. **C - Completo e Total.** 3^a ed. São Paulo: MAKRON Books, 1996.
- TAN, W. M. **Developing USB PC Peripherals. Using the Intel 8x930Ax USB Microcontroller.** San Diego: Annabooks, 1997.
- PEREIRA, F. **Tecnologia ARM: Microcontroladores de 32 Bits.** 1^a ed. São Paulo: Editora Érica, 2007.

Manuais

- MICROCHIP Technology. AN1164 - USB CDC Class on an Embedded Device (DS01164A). 2008.
- MICROCHIP Technology. Compiled Tips 'N Tricks Guide (DS01146B). USA, 2009.
- MICROCHIP Technology. Data Sheet PIC18F2331/2431/4331/4431 (DS39616B). USA, 2003.
- MICROCHIP Technology. Data Sheet PIC18F2420/2520/4420/4520 (DS39631A). USA, 2004.
- MICROCHIP Technology. Data Sheet PIC18F2455/2550/4455/4550 (DS39632B). USA, 2004.
- MICROCHIP Technology. Data Sheet PIC18F2480/2580/4480/4580 (DS39637C). USA, 2007.
- MICROCHIP Technology. Data Sheet PIC18FXX2 (DS39564B). USA, 2002.
- MICROCHIP Technology. Device Configuration Overview, and Design Tips for the PICmicro® Microcontroller Configuration. 2001.
- MICROCHIP Technology. Flash Microcontroller Programming Specification (DS39622K). USA, 2007.
- MICROCHIP Technology. Getting Started - I2C™ Master Mode, Version 0.4. 2001.
- MICROCHIP Technology. Getting Started - Oscillator Overview, Design Tips and Troubleshooting of the PICmicro® Microcontroller Oscillator. 2001.
- MICROCHIP Technology. Getting Started - Power Considerations. 2002.
- MICROCHIP Technology. MPLAB® C18 C Compiler Getting Started (DS51295F). USA, 2005.
- MICROCHIP Technology. MPLAB® C18 C Compiler Libraries (DS51297F). USA, 2005.
- MICROCHIP Technology. MPLAB® C18 C Compiler User's Guide (DS51288J). USA, 2005.
- MICROCHIP Technology. MPLAB® IDE User's Guide with MPLAB Editor and MPLAB SIM Simulator (DS51519C). USA, 2009.
- MICROCHIP Technology. PICmicro™ Mid-Range MCU Family Reference Manual (DS33023A). USA. 1997.
- MICROCHIP Technology. Reset Causes and Effects. 2002.
- Philips Semiconductors. The I 2C-Bus Specification, Version 2.1. 2000.
- SANDISK CORPORATION. SanDisk Secure Digital (SD) Card Product Manual, Rev. 1.9, 2003.
- Universal Serial Bus 2.0 Specifications, Rev. 2.0. USA, 2000.

Universal Serial Bus 3.0 Specifications, Rev. 1.0. USA, 2008.

Universal Serial Bus Language Identifiers (LANGIDs), Version 1.0. USA, 2000.

Sites

<http://en.wikipedia.org/wiki/Microcontroller>

http://en.wikipedia.org/wiki/PIC_microcontroller

<http://en.wikipedia.org/wiki/Uart>

<http://en.wikipedia.org/wiki/usb>

<http://ww1.microchip.com/downloads/en/devicedoc/spi.pdf>

http://www.camiresearch.com/Data_Com_Basics/RS232_standard.html

<http://www.esacademy.com/faq/i2c/index.htm>

<http://www.usb.org/about>

http://www.usb.org/developers/defined_class

Marcas Registradas

PIC® MCU, dsPIC®, MPLAB® IDE, MPLAB® ICD2, PIC START®, PROMATE® são marcas registradas da Microchip Technology Inc. nos EUA e em outros países.

REAL ICE™, ICSP™, In-Circuit Serial Programming™, MPSIM™ são marcas da Microchip Technology Inc. nos EUA e em outros países.

Todos os demais nomes registrados, marcas registradas ou direitos de uso citados neste livro pertencem aos respectivos proprietários.

Índice Remissivo

- A**
- ACK, 293
 - AckI2C, 294
 - Acknowledge, 293
 - Acos, 113
 - Access RAM, 144
 - Alimentação do circuito, 161
 - Arquitetura, 137
 - Arquivos de cabeçalho, 166
 - asin, 113
 - Assembly, 120
 - atan, 113
 - Atob, 92
 - Atof, 92
 - Atoi, 92
 - Atol, 92
 - Auto, 43, 47
 - Auto-wake-up, 211
- B**
- Base numérica, 49
 - Baud Rate, 212, 293
 - BaudUSART, 215
 - BGR, 212
 - Bit de ativação, 197
 - prioridade, 197
 - sinalização, 197
 - BOR, 159
 - Break, 61
 - Brown-out Reset, 159
 - BSR, 144
 - Btoa, 93
 - Bulk Data Transfers, 355
 - BusyADC, 270
 - BusyUSART, 216
- C**
- Capacitores de desacoplamento, 161
 - Capture, 243
 - Caractere especial, 187
 - CCP, 243
 - CDCInitEP, 386
 - CDCTxService, 388
 - Ceil, 113
 - CGRAM, 187
 - Circuito de Reset, 158
 - Classes, 369
 - ClearCSSWSPI, 315
 - Clock_test, 299
 - CloseADC, 270
 - CloseCapture, 252
 - CloseCompare, 254
 - CloseI2C, 296
 - ClosePWM, 255
- D**
- Dados Numéricos, 49
 - Data Latch, 175, 177
 - Data Toggle, 351
 - DataRdyI2C, 297
 - DataRdySPI, 312
 - DataRdyUSART, 216
 - DDRAM, 185
 - dead-band delay, 248
 - dead_delay, 248
 - Declaração, 49
 - Delay1TCY, 130
 - Desritores, 362
 - Diretivas, 66
 - Display LCD, 183, 190
 - Do-while, 59
 - Duty cycle, 244
- E**
- EC, 156
 - ECCP, 245
 - Else, 57
 - Enable bit, 197
 - Encapsulamento, 165
 - Endpoint, 347
 - Enum, 63
 - Enumerações, 63
 - EEPROM interna, 132
 - EPROM, 22
 - escreve_bloco, 338
 - escreve_mem_EEPROM, 132
 - I/O, 174
 - I²C, 291-293, 298
 - ICSP, 164
 - Idata, 71
 - Identificador, 44
 - Idle, 157
 - IdleI2C, 297
 - leetomchp, 116
 - If, 57
 - imprime_buffer_lcd, 190
 - imprime_string_lcd, 190
 - inicia_sd, 338
 - Instrução, 48
 - estendida, 163
 - Interrupção, 195
 - com nível de prioridade, 196
- F**
- fabs, 115
 - Flag bit, 197
 - Flash, 22
 - interna, 133
 - Floor, 113
 - Fmod, 115
 - For, 60
 - Fprintf, 81
 - Fputs, 78
 - Frame, 349
 - Frexp, 116
 - FSCM, 163
 - Função, 55
 - principal, 55
 - secundária, 55
- G**
- getI2C, 296
 - getUART, 223
 - getUSART, 219
 - getsI2C, 296
 - getsSPI, 312
 - getsUART, 223
 - getsUSART, 219
 - getsUSBUSART, 387
 - Goto, 62
 - GPRs, 144
- H**
- _H_USART, 76
 - _H_USER, 76
 - Halt, 360
 - High priority level, 195
 - HLVD, 163
 - HS, 155
- I**
- I/O, 174
 - I²C, 291-293, 298
 - ICSP, 164
 - Idata, 71
 - Identificador, 44
 - Idle, 157
 - IdleI2C, 297
 - leetomchp, 116
 - If, 57
 - imprime_buffer_lcd, 190
 - imprime_string_lcd, 190
 - inicia_sd, 338
 - Instrução, 48
 - estendida, 163
 - Interrupção, 195
 - com nível de prioridade, 196
- J**
- sem nível de prioridade, 197
 - intRC, 156
 - Isalnum, 87
 - Isalpha, 88
 - IsBOR, 129
 - Isctrl, 90
 - Isdigit, 88
 - Isgraph, 91
 - Islower, 88
 - IsLVD, 129
 - IsMCLR, 129
 - IsPOR, 130
 - Isprint, 91
 - Ispunct, 92
 - ISR, 201
 - IsSpace, 89
 - Isupper, 89
 - IsWDTTO, 130
 - IsWDTWU, 130
 - IsWU, 130
 - Isxdigit, 89
 - Ltoa, 93
- L**
- LAT, 177
 - lcd_cursor_home, 190
 - lcd_desloca_cursor, 190
 - lcd_desloca_mensagem, 190
 - lcd_escreve_dado, 190
 - lcd_inicia, 190
 - lcd_LD_cursor, 190
 - lcd_le_dado, 190
 - lcd_limpa_tela, 190
 - lcd_posicao, 190
 - lcd_status, 190
 - Ldexp, 117
 - le_bloco, 338
 - le_mem_EEPROM, 132
 - le_mem_flash, 133
 - le_seq_bloco, 338
 - Linha de comando, 36
 - Log, 117
 - Low
 - power, 154
 - priority level, 195
 - Ltoa, 93
 - LVP, 164
- M**
- Macros Predefinidas, 74
 - Main, 55
 - Manipulação de caracteres, 87
 - Matriz, 50
 - multidimensional, 51
 - unidimensional, 50
 - Mchptoeiee, 116

- M**
- Memchr, 94
 - Memchrgpm, 94
 - Memcmp, 95
 - Memcpy, 96
 - Memmove, 97
 - Memória de dados, 29, 144
 - programa, 29, 150
 - Memset, 97
 - Métodos de programação, 164
 - Microcontrolador, 21
 - Microframe, 349
 - Modf, 118
 - Modo estendido, 43
 - MPLAB[®] C18, 24, 26, 35, 44
 - MPLAB[®] IDE, 24, 35, 77
 - Multi-Master, 291
- N**
- Nop, 131
 - NotAckI2C, 294
 - NRZI, 347
- O**
- OpenADC, 271
 - OpenCapture, 252, 254
 - OpenCompare, 254
 - OpenI2C, 295
 - OpenPWM, 255
 - OpenSPI, 311
 - OpenSWSPI, 315
 - OpenTimer, 230-233
 - OpenUART, 222
 - OpenUSART, 213
 - Operadores, 51
 - aritméticos, 52
 - bit a bit, 52
 - de atribuição, 51
 - lógicos, 54
 - relacionais, 53
 - Oscilador, 152
 - low-power, 229
 - Oscillator Start-up Timer, 160
 - OST, 160
 - Otimização do código, 37
 - OTP, 22
 - Overlay, 43, 48
- P**
- Palavras-chaves, 45
 - PID, 349
 - Pira, 151
 - Pipe, 348
 - PIPELINE, 138
 - PLL Lock Time-out, 160
 - Ponteiro, 47, 54
 - POR, 159
 - PORT, 176
- Postscaler**, 228
- Pow**, 118
- Power-on Reset**, 159
- Power-up Timer**, 160
- Prescaler**, 228
- Printf**, 78
- Priority bit**, 197
- Program Counter**, 150
- Proteção do código**, 152
- Protocolo RS-232**, 209
- Pull-ups internos**, 179
- Pt.c.** 77
- putcI2C**, 297
- putcSPI**, 313
- putcSWSPI**, 315
- putcUART**, 222
- putcUSART**, 217
- putrsUSART**, 219
- putrsUSBUSART**, 388
- puts**, 77
- putsI2C**, 298
- putsSPI**, 313
- putsUART**, 222
- putsUSART**, 218
- putsUSBUSART**, 388
- putUSBUSART**, 387
- PWM**, 243
- PWRT**, 160
- R**
- Ram**
 - far, 46
 - near, 46
- Rand**, 119
- ReadADC**, 277
- ReadCapture**, 253
- ReadI2C**, 296
- ReadSPI**, 312
- ReadTimer**, 234
- ReadUART**, 223
- ReadUSART**, 219
- Register**, 48
- Remote wake-up**, 360
- Reset**, 131, 157
- RestartI2C**, 294
- Return**, 62
- RISC**, 137
- Ricf**, 85
- Rlncf**, 84
- Rom**, 22
 - far, 46
 - near, 46
- Romdata**, 71
- Rótulo**, 62
- Rrcf**, 85
- Rnfcf**, 84
- Run**, 157
- S**
- SD Card**, 323, 324, 330, 338
- Sectiontype**, 70
- Serviço de interrupção**, 201
- SetChanADC**, 277
- SetCSSWSPI**, 315
- SetDCPWM**, 256
- SetOutputPWM**, 257
- SetTmrCPSrc**, 234
- SFRs**, 144
- SIE**, 377
- Sin**, 114
- Sinh**, 114
- Sleep**, 132, 157
- Software reset**
 - instruction, 159
- SPI**, 308, 310, 314
- sprintf**, 81
- sqrt**, 119
- srand**, 120
- Stack full or underflow**
 - reset, 159
- Stack**, 151
- StartI2C**, 294
- Static**, 43, 47
- Status register**, 162
- stderr**, 76
- stdout**, 76
- StopI2C**, 295
- strcat**, 98
- strchr**, 97
- strcmp**, 100
- strcpy**, 101
- strcspn**, 102
- strlen**, 103
- strlwr**, 103
- strncat**, 104
- strncmp**, 105
- strncpy**, 106
- strpbk**, 107
- strrchr**, 107
- strspn**, 108
- strstr**, 109
- strtok**, 110
- struct**, 64
- strupr**, 111
- SWAckI2C**, 300
- Swapf**, 86
- SWGetI2C**, 301
- SWGtsI2C**, 301
- Switch**, 57
- SWNotAckI2C**, 300
- SWPutI2C**, 301
- SWPutI2C**, 302
- SWReadI2C**, 301
- SWRestartI2C**, 300
- SWStartI2C**, 300
- SWStopI2C**, 300
- SWWriteI2C**, 301
- T**
- tan**, 114
- tanh**, 114
- Temporizadores**, 228
- Test mode**, 361
- Tipos de dados**, 45
 - definidos, 65
- Tmpdata**, 72
- Tolower**, 90
- Toupper**, 90
- TRIS**, 175
- Two-Speed Start-up**, 161
- typedef**, 65
- U**
- _usart_putc**, 83
- _user_putc**, 83
- UART**, 221
- Udata**, 71
- ULA**, 143
- Ultos**, 93
- Uniões**, 66
- Union**, 66
- USART**, 209
- USB**, 157, 344, 346, 348, 375, 377
- USBCheckCDCRequest**, 386
- USBDeviceAttach**, 384
- USBDeviceDetach**, 384
- USBDeviceInit**, 381
- USBDeviceTasks**, 381
- USBEnableEndpoint**, 383
- USBStallEndpoint**, 383
- USBTransferOnePacket**, 383
- V**
- Variável**
 - global, 49
 - local, 49
- Varlocate**, 73
- Vfprintf**, 82
- Volatile**, 46
- Vprintf**, 81
- Vsprintf**, 83
- W**
- Watchdog Timer**, 130, 157-159, 235
- WDT**, 157, 159, 235
- While**, 59
- Work Register**, 143
- WREG**, 143
- WriteI2C**, 297
- WriteSPI**, 313
- WriteSWSPI**, 315
- WriteTimer**, 234
- WriteUART**, 222
- WriteUSART**, 217
- WUE**, 211
- X**
- XT**, 155