Microcontroladores PIC
Programação em C

Seja Nosso 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

Ao preencher e remeter a ficha de cadastro constante no final desta publicação, você passará a receber, automaticamente, 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) 6197.4060
3. Carta: Rua São Gil, 159 - Tatuapé - CEP 03401-030 - São Paulo - SP

Invista em você.
Visite uma livraria.

Fábio Pereira

Microcontroladores PIC
Programação em C

Editora Érica Ltda.
2003 - 2ª Edição

Conselho Editorial:

Diretor Editorial:	Antonio Marco Vicari Cipelli
Diretor Comercial:	Paulo Roberto Alves
Diretor de Publicidade:	Waldir João Sandrini
Capa:	Maurício S. de França
Editoração:	Rosana Ap. Alves dos Santos
Revisão Interna:	Érica Regina Pagano
Revisão Gramatical:	Marlene Teresa Santin Alves
Desenhos:	Pedro Paulo V. Herruzo
	Flávio Eugênio de Lima
Coordenação e Revisão:	Rosana Arruda da Silva

Copyright © 2003 da Editora Érica Ltda.

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

Pereira, Fábio, 1974-
Microcontroladores PIC: Programação em C / Fábio Pereira. -- São Paulo: Érica, 2003.

Bibliografia.
ISBN 85-7194-935-2

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

02-6151 CDD-005.133

Índices para catálogo sistemático

1. C: Linguagem de programação: Aplicações com microcontroladores PIC: Computadores: Processamento de dados 005.133

Todos os direitos reservados. Proibida a reprodução total ou parcial, por qualquer meio ou processo, especialmente por sistemas gráficos, microfílmicos, fotográficos, reprográficos, fonográficos, videográficos, internet, e-books. Vedada a memorização e/ou recuperação total ou parcial em qualquer sistema de processamento de dados e a inclusão de qualquer parte da obra em qualquer programa juscibernético. Essas proibições aplicam-se também às características gráficas da obra e à sua editoração. A violação dos direitos autorais é punível como crime (art. 184 e parágrafos, do Código Penal, cf. Lei nº 6.895, de 17.12.80) com pena de prisão e multa, conjuntamente com busca e apreensão e indenizações diversas (artigos 102, 103 parágrafo único, 104, 105, 106 e 107 itens 1, 2 e 3 da Lei nº 9.610, de 19/06/98, Lei dos Direitos Autorais).

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

"Algumas imagens utilizadas neste livro foram obtidas a partir do CorelDRAW 7, 8 e 9 e da Coleção do MasterClips/ MasterPhotos© da IMSI, 1985 Francisco Blvd. East, San Rafael, CA 94901-5506, USA."

Editora Érica Ltda.

Rua São Gil, 159 - Tatuapé

CEP: 03401-030 - São Paulo - SP

Fone: (11) 295-3066 - Fax: (11) 6197-4060

www.editoraerica.com.br

Requisitos de Hardware e de Software

Hardware:

- Microcomputador Pentium 200 MHz ou superior;
- 16 MB de memória RAM;
- 50 Mbytes livres de HD;
- Acesso a Internet.

Software:

- Sistema operacional Windows 95 ou superior;
- Compilador CCS, PCWH, PCM, PCH ou PCW;
- Ambiente MPLAB;
- IC-PROG ou um software de programação.

Sobre o Autor

Nascido em São Francisco do Sul, Fábio Pereira é formado em direito e em eletrônica, e reside em Joinvile - SC.

Atua desde o início da década de 90 na área de software e hardware. Desenvolveu diversos sistemas em plataformas, como: PCs, microprocessadores Z-80, 8086, 80386, etc. e microcontroladores 8051 e PICs. É autor do livro Microcontroladores PIC: Técnicas Avançadas.

Além de atuar na área de projetos industriais e automotivos (na qual já desenvolveu diversos projetos comerciais, incluindo tacômetros digitais de alta precisão e equipamentos de controle e medição industrial), leciona as disciplinas de linguagem C e microcontroladores no SENAI-SC, no qual também auxilia no desenvolvimento de projetos didáticos envolvendo microcontroladores.

É também sócio da SCtec, empresa que atua na área de desenvolvimento de projetos e sistemas de automação industrial (www.sctec.com.br).

Dedicatória

À minha esposa Josi, aos meus pais e à minha avó.

Tu me ensinarás o caminho da vida; em tua presença há plenitude de alegria, à tua direita delícias eternas.

Sl 16, 11

Agradecimentos

Agradeço a todos aqueles que colaboraram direta ou indiretamente para que este livro pudesse ser escrito, e em especial:

- A André Rabner, da Artimar, pelas amostras e material de consulta;
- A CCS pela cópia do compilador PCWH e pelo suporte fornecido;
- Ao pessoal da Editora Érica por acreditar no meu trabalho;
- Aos meus leitores e críticos que contribuem de uma forma ou de outra para a melhoria dos meus livros.

Prefácio

Existem diversas literaturas sobre a linguagem C disponíveis no mercado, mas quando procuramos um livro que seja voltado para as aplicações com microcontroladores, este universo reduz-se drasticamente.

Foi para atender a esta lacuna que surgiu este livro, fruto de um extenso trabalho de pesquisa e com uma abordagem simples e didática.

Partindo dos princípios gerais de programação, até os detalhes mais profundos e complexos da linguagem C, sempre enfatizando os microcontroladores PIC, utilizando diversos exemplos e exercícios.

São também apresentadas as diretivas e funções internas dos compiladores CCS, os mapas de memória dos PICs utilizados no livro (12F675, 16F628, 16F876, 16F877, 18F252 e 18F452) e os conjuntos de instruções assembly de 14 e 16 bits.

Além disso, foi dedicado um capítulo do livro ao estudo de técnicas de programação, abrangendo otimização de código, interrupções, interfaceamento (teclados e displays LCD), comunicação serial (assíncrona, SPI, I^2C, 1-Wire, além de uma visão geral sobre CAN, LIN, etc.), geração de PWM, medição de temperatura com o chip Dallas 18S20, conversão analógico/digital, dentre outros.

Trata-se de uma obra indispensável a todos aqueles que desejam tanto conhecer a linguagem C em maior profundidade, quanto a sua aplicação aos microcontroladores e pequenos sistemas em geral.

Sobre o Material Disponível na Internet

O material disponível na Internet contém os arquivos de exemplo e bibliotecas apresentados no capítulo 12 e arquivos de cabeçalho com as definições de registradores utilizadas nos programas.

Para utilizá-los, é necessário possuir instalados em sua máquina o compilador PCWH (para utilizar os PICs das séries 12,16 e 18), o PCW (para utilizar os PICs das séries 12 e 16), ou ainda os compiladores PCM ou PCH, além do ambiente MPLAB (para realização de simulações) e um software de programação (no caso do autor, o IC-PROG).

Arquivos.EXE	48 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:

1. Por meio do módulo pesquisa. Localize o livro desejado, digitando palavras-chaves (nome do livro ou do autor). Aparecerão os dados do livro e o arquivo para download, então dê um clique sobre o arquivo executável que será transferido.

2. Por meio do botão "Download". Na página principal do site, clique no item "Download". Será exibido um campo, no qual devem ser digitadas palavras-chaves (nome do livro ou do autor). Serão exibidos o nome do livro e o arquivo para download. Dê um clique sobre o arquivo executável que será transferido.

Procedimento para Descompactação

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

C:\Microcontroladores PIC

Segundo passo: prossiga com a instalação, clicando no botão Unzip, o qual se encarrega de descompactar os arquivos. Logo abaixo dessa tela, aparece a barra de status a qual monitora o processo para que você acompanhe. Após o término, outra tela de informação surgirá, 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.

Índice Analítico

Capítulo 1 - Introdução 15
1.1. Um Pouco de História 16
1.2. Programação de PICs em C 18
1.3. Mensagem aos Programadores Pascal 19
1.4. Convenções Deste Livro 19
1.5. Requisitos de Hardware e Software 20

Capítulo 2 - Princípios de Programação 23
2.1. Fluxogramas 25
2.2. Álgebra Booleana 27
2.3. Variáveis e Dados 29
2.4. Operadores 30
2.5. Exercícios 31

Capítulo 3 - Compilador CCS C 33
3.1. Ambiente Integrado de Desenvolvimento (IDE) 34
3.2. Integração com o MPLAB 43

Capítulo 4 - Introdução à Linguagem C 47
4.1. Palavras Reservadas da Linguagem 53
4.2. Identificadores 53
4.3. Comandos e Funções Importantes 54
4.3.1. Comando If 54
4.3.2. Comando While 55
4.3.3. Função Printf 55
4.3.4. Função GETC 56

Capítulo 5 - Variáveis e Tipos de Dados 57
5.1. Modificadores de Tipo 58
5.2. Outros Tipos Específicos do Compilador CCS C 59
5.3. Declaração de Variáveis 61
5.3.1. Variáveis Locais 64
5.4. Constantes 65
5.4.1. Códigos de Barra Invertida 66
5.4.2. Constantes Binárias, Hexadecimais e Octais 67
5.5. Operadores 67
5.6. Expressões 67
5.7. Conversão de Tipos 69
5.8. Modificadores de Acesso 73
5.9. Modificadores de Armazenamento 73
5.10. Alocação de Memória 74
5.11. Redefinindo Tipos de Dados 75
5.12. Exercícios 76

Capítulo 6 - Operadores 79
6.1. Atribuição 79
6.2. Aritméticos 80
6.3. Relacionais 81
6.4. Lógicos Booleanos 82
6.5. Lógicos Bit a Bit 83
6.5.1. Operador & (And) 83
6.5.2. Operador | (OR) 84
6.5.3. Operador ^ (XOR) 84
6.5.4. Operador ~ (NOT) 85
6.5.5. Operadores de Deslocamento << E >> 85
6.6. Memória 87
6.6.1. Operador & 87
6.6.2. Operador * 87
6.7. Outros Operadores 88
6.7.1. Operador? 88
6.7.2. Operador Vírgula 89
6.7.3. Operador Ponto 90
6.7.4. Operador -> 90
6.7.5. Operador de Modelagem de Tipo 90
6.7.6. Operador Sizeof 91
6.8. Associação de Operadores 91
6.9. Precedência dos Operadores 92
6.10. Exercícios 93

Capítulo 7 - Declarações de Controle 95
7.1. Comando If 95
7.2. Comando Switch 98
7.2.1. Cláusula Break 100
7.3. Estruturas de Repetição 101
7.3.1. Laço For 101
7.3.2. Laço While 106
7.3.3. Laço Do-While 108
7.4. Comando Goto 109
7.5. Exercícios 110

Capítulo 8 - Tipos e Dados Avançados 113
8.1. Ponteiros 113
8.1.1. Operações com Ponteiros 115
8.1.2. Tópicos Importantes Sobre Ponteiros 117
8.2. Matrizes de Dados 117
8.2.1. Matrizes Multidimensionais 121
8.2.2. Strings de Caracteres 122
8.2.3. Matrizes e Ponteiros 123
8.3. Estruturas de Dados 124
8.3.1. Operações com Estruturas 125

8.3.2. Ponteiros para Estruturas 127
8.3.3. Campos de Bit 128
8.4. Uniões 130
8.5. Enumerações 132
8.6. Streams 133
8.7. Exercícios 134

Capítulo 9 - Funções 137
9.1. Forma Geral 137
9.2. Regras de Escopo 139
9.3. Passagem de Parâmetros 140
9.4. Matrizes com Argumentos de uma Função 142
9.5. Estruturas como Argumentos de uma Função 144
9.6. Funções com Número de Parâmetros Variável 144
9.7. Retorno de Valores 145
9.8. Retorno de Valores em Funções Assembly 147
9.9. Protótipos de Função 148
9.10. Recursividade 149
9.11. Exercícios 149

Capítulo 10 - Diretivas do Compilador 151

Capítulo 11 - Funções do Compilador 175
11.1. Matemáticas 175
11.2. Manipulação de Caracteres 183
11.3. Memória 193
11.4. Atraso 195
11.5. Manipulação de Bit / Byte 196
11.6. Entrada / Saída 202
11.7. Analógicas 206
11.8. Manipulação de Timers 211
11.9. Comparação / Captura / PWM 218
11.10. Manipulação da EEPROM Interna 219
11.11. Controle do Processador 222
11.12. Porta Paralela Escrava 228
11.13. Comunicação Serial Assíncrona 230
11.14. Comunicação I^2C 240
11.15. Comunicação SPI 243

Capítulo 12 - Tópicos Avançados 247
12.1. Escrevendo Código Eficiente em C 247
12.1.1. Utilização de Variáveis Booleanas 247
12.1.2. Testes Condicionais 249
12.1.3. Rotinas Matemáticas 250
12.1.4. Utilizando Assembly no Código C 251

12.2. Entrada e Saída em C 251
12.2.1. Modo Padrão 252
12.2.2. Modo Fixo 253
12.2.3. Modo Rápido 254
12.2.4. Outra Modalidade de Acesso 254
12.3. Interrupções em C 254
12.3.1. Tratamento "Automático" 255
12.3.2. Tratamento "Manual" 256
12.3.3. Priorização de Interrupções 257
12.4. Utilizando os Timers Internos 258
12.4.1. Timer 0 258
12.4.2. Timer 1 259
12.4.3. Obtendo Mais Precisão dos Timers 260
12.5. Comunicando-se com o PIC 262
12.5.1. Comunicação Serial Assíncrona 263
12.5.2. Protocolo SPI 268
12.5.3. Protocolo I^2C 274
12.5.4. Protocolo 1-WIRE 282
12.5.5. LIN 291
12.5.6. Protocolo CAN 296
12.5.7. Interfaces Elétricas 297
12.6. Leitura de Teclas e de Teclados 300
12.7. Apresentação em Display 307
12.7.1. Displays LED de Sete Segmentos 308
12.7.2. Módulo LCD 310
12.8. Leitura de Tensões Analógicas com o PIC 318
12.8.1. Conversor A/D Interno 319
12.8.2. Conversor A/D Delta - Sigma 322
12.9. Módulo PWM 326

Apêndice A - Tabela ASCII 329

Apêndice B - Funções C - Referência Rápida 331

Apêndice C - Instruções PIC 335

Apêndice D - Mapas de Memória 339

Apêndice E - Respostas dos Exercícios 351

Índice Remissivo 353

Referências Bibliográficas 357

Capítulo 1

Introdução

A criação de programas para microcontroladores e microprocessadores pode ser uma tarefa desgastante à medida que aumenta a complexidade da aplicação sendo desenvolvida.

Os primeiros dispositivos programáveis tinham seus programas escritos com códigos chamados códigos de máquina, que consistiam normalmente em dígitos binários que eram inseridos por meio de um dispositivo de entrada de dados (teclado, leitora de cartões, fitas perfuradas ou discos magnéticos) para então serem executados pela máquina.

Desnecessário dizer que a programação em código de máquina era extremamente complexa, o que implicava em um elevado custo, além de muito tempo para o desenvolvimento de uma aplicação.

Diante da necessidade crescente de programação de sistemas, foi natural o surgimento de uma nova forma de programação de sistemas. Esta foi a origem da linguagem *Assembly*.

Assembly consiste em uma forma alternativa de representação dos códigos de máquina usando mnemônicos, ou seja, abreviações de termos usuais que descrevem a operação efetuada pelo comando em código de máquina. A conversão dos mnemônicos em códigos binários executáveis pela máquina é feita por um tipo de programa chamado *Assembler* (montador).

Assim, em vez de escrevermos o comando em código de máquina 0011000010001100, o programador poderia simplesmente utilizar o comando MOVLW 0x8C para realizar a mesma tarefa.

Sem dúvida nenhuma, a representação em *Assembly* da instrução é muito mais simples do que aquela utilizando código de máquina, no entanto a utilização do *Assembly* não resolveu todos os problemas dos programadores.

A linguagem *Assembly* é de baixo nível, ou seja, não possui nenhum comando, instrução ou função além daqueles definidos no conjunto de instru-

ções do processador utilizado. Isto implica em um trabalho extra do programador para desenvolver rotinas e operação que não fazem parte do conjunto de instruções do processador, produzindo, por conseguinte, programas muito extensos e complexos com um fluxo muitas vezes difícil de ser seguido.

É aí que entram as chamadas linguagens de alto nível. Elas são criadas para permitir a programação utilizando comandos de alto nível e que são posteriormente traduzidos para a linguagem de baixo nível (*assembly* ou diretamente para código de máquina) do processador utilizado.

1.1. Um Pouco de História

Como já vimos, as linguagens de baixo nível, surgidas praticamente junto com a era dos computadores digitais, possuem diversos aspectos negativos, como a complexidade, falta de compatibilidade entre diferentes sistemas, pouca legibilidade.

A necessidade de uma abordagem mais concisa e simplificada para comandar computadores surgiu naturalmente com a evolução e utilização dos computadores digitais. Essa abordagem simplificada, mais próxima da forma humana de pensamento, é chamada de linguagem de alto nível.

A primeira linguagem de alto nível com real aceitação pela comunidade de programadores foi FORTRAN (abreviação de ***FOR**mula **TRAN**slator*, ou tradutor de fórmulas), uma linguagem voltada para a análise e resolução de problemas matemáticos criada na metade da década de 50, por pesquisadores da IBM.

Em seguida vieram linguagens como COBOL (abreviação *de **CO**mmon **B**usiness **O**riented **L**anguage*, ou linguagem comum para aplicações comerciais) em 1959, voltada para o desenvolvimento de aplicações comerciais, ALGOL (***ALGO**rithmic **L**anguage*, ou linguagem algorítmica) em 1960, linguagem de programação genérica e poderosa que originou diversas outras linguagens de alto nível como PASCAL e C.

A linguagem C, por sua vez, foi criada em 1972, por Dennis Ritchie, da Bell Laboratories, e consiste, na realidade, em uma linguagem de nível intermediário entre o Assembly e as linguagens de alto nível.

É uma linguagem de programação genérica desenvolvida para ser tão eficiente e rápida quanto a linguagem Assembly e tão estruturada e lógica quanto as linguagens de alto nível (PASCAL, JAVA, etc.).

Suas origens são atribuídas a três linguagens de programação:

- ALGOL;
- BCPL (***B**asic **C**ombined **P**rogramming **L**anguage*: linguagem de programação básica combinada) - derivada da linguagem CPL (que, por sua

vez, foi derivada da ALGOL), foi desenvolvida em 1969, e alia a estrutura da ALGOL à eficiência do Assembly, no entanto BCPL era uma linguagem demasiado complexa e relativamente limitada, razão pela qual não teve êxito;

- B: linguagem desenvolvida por Ken Thompson da Bell Laboratories, em 1970. Baseada na linguagem CPL, B foi uma tentativa de simplificar e facilitar a linguagem CPL, no entanto também a B não obteve êxito devido às sérias limitações que impunha ao programador.

Até o desenvolvimento do C, não existiam linguagens de programação de alto nível adequadas à tarefa de criação de sistemas operacionais (programas especiais utilizados para o controle genérico de um computador) e outros softwares de baixo nível, restando aos desenvolvedores utilizar o *Assembly* para a execução destas tarefas.

No entanto, o *Assembly* apresenta prós e contras em sua utilização. Vejamos alguns deles:

Prós:

- Eficiência: devido à proximidade com o hardware da máquina, o *Assembly* é sem dúvida uma linguagem extremamente eficiente (desde que o programador saiba o que está fazendo !);
- Velocidade: devido à sua grande eficiência, os programas em *Assembly* são também muito mais rápidos que os criados em outras linguagens.

Contras:

- Complexidade: o *Assembly* necessita que o programador possua um profundo conhecimento do hardware utilizado. Além disso, somente estão disponíveis os comandos do processador utilizado. Qualquer operação mais complexa deve ser traduzida em um conjunto de comandos *Assembly;*
- Portabilidade: programas *Assembly*, que pela sua natureza não são portáveis. Isto significa que para utilizar um programa *Assembly* de um sistema em outro diferente, o programador deve executar a tradução completa do programa para o sistema de destino.

Foi principalmente a partir das necessidades de reescrita do sistema operacional UNIX (que até então era escrito em *Assembly*) que surgiu a linguagem C.

De fato, a implementação da linguagem é tão poderosa que C foi a escolhida para o desenvolvimento de outros sistemas operacionais além do UNIX, como o WINDOWS e o LINUX.

Assim como outras linguagens de alto nível, C utiliza a filosofia de programação estruturada, ou seja, os programas são divididos em módulos ou estruturas (que em C são chamadas de funções) independentes entre si e com o objetivo de realizar determinada tarefa.

Desta forma, a programação estruturada permite uma construção mais simples e clara do software de aplicação, o que permite a criação de programas de maior complexidade (quando comparada a outras linguagens não estruturadas como *Assembly* ou BASIC).

1.2. Programação de PICs em C

A utilização de C para a programação de microcontroladores com os PICs parece uma escolha natural e realmente é.

Atualmente, a maioria dos microcontroladores disponíveis no mercado contam com compiladores de linguagem C para o desenvolvimento de software.

O uso de C permite a construção de programas e aplicações muito mais complexas do que seria viável utilizando apenas o *Assembly*.

Além disso, o desenvolvimento em C permite uma grande velocidade na criação de novos projetos, devido às facilidades de programação oferecidas pela linguagem e também à sua portabilidade, o que permite adaptar programas de um sistema para outro com um mínimo esforço.

Outro aspecto favorável da utilização da linguagem C é a sua eficiência.

Eficiência no jargão dos compiladores é a medida do grau de inteligência com que o compilador traduz um programa em C para o código de máquina. Quanto menor e mais rápido o código gerado, maior será a eficiência da linguagem e do compilador.

Conforme já dissemos, C, devido a sua proximidade com o hardware e o *Assembly*, é uma linguagem extremamente eficiente. De fato, C é considerada como a linguagem de alto nível mais eficiente atualmente disponível.

Repare que o aspecto eficiência é realmente muito importante quando tratamos de microcontroladores cujos recursos são tão limitados como nos PICs, afinal, quando dispomos de apenas 512 palavras de memória de programa e 25 bytes de RAM (como no PIC12C508 e 16C54), é imprescindível que se economize memória!

Além disso, a utilização de uma linguagem de alto nível como C permite que o programador preocupe-se mais com a programação da aplicação em si, já que o compilador assume para si tarefas como o controle e localização das variáveis, operações matemáticas e lógicas, verificação de bancos de memória, etc.

1.3. Mensagem aos Programadores Pascal

Alguns dos leitores deste livro, assim como o autor, podem Ter experiência de programação com outras linguagens como BASIC, Pascal, etc.

Especificamente em relação à linguagem Pascal, acreditamos que o aprendizado de C será feito sem maiores problemas.

De fato, C e Pascal são linguagens muito parecidas, sendo as principais diferenças:

- Bloco de comandos: em Pascal, um bloco de comandos é delimitado pelas palavras **begin** e **end**, em C pelos caracteres { e };
- O operador de atribuição em Pascal é o:=, em C é apenas =;
- O operador relacional de igualdade em Pascal é o =, em C é o ==;
- Em Pascal, temos as funções (**function**) e procedimentos (**procedure**), sendo que um procedimento pode ser interpretado como uma função que não retorna valores. Em C, temos apenas as funções, que não recebem identificador especial e podem ou não retornar resultados;
- Pascal é uma linguagem mais conservadora e rígida com uma forte checagem de tipos e variáveis, o que restringe em muitos casos a liberdade do programador. C, ao contrário, é uma linguagem altamente liberal, ou seja, permite ao programador realizar praticamente qualquer coisa, inclusive erros clássicos de programação;
- A quantidade de operadores disponíveis em Pascal é menor que os disponíveis em C;
- Pascal é uma linguagem menos eficiente que C, porém é mais simples e de leitura mais agradável que C.

Estas são apenas algumas das diferenças existentes entre C e Pascal. De qualquer forma, o leitor poderá, por si mesmo, verificar as semelhanças e diferenças existentes entre a linguagem C e as demais linguagens existentes na atualidade.

1.4. Convenções Deste Livro

Antes de iniciar os estudos sobre o compilador C, é necessário esclarecer ao leitor as convenções adotadas neste livro.

Os termos em língua inglesa serão sempre escritos em caracteres itálicos.

As referências a nomes de funções e palavras reservadas da linguagem C serão sempre grafadas em negrito.

Os operandos, termos ou parâmetros opcionais de uma declaração estarão sempre grafados em itálico.

1.5. Requisitos de Hardware e Software

Os programas demonstrados neste livro foram testados com a versão 3.127 do compilador CCS e a versão 5.7 do ambiente integrado MPLAB.

Os exemplos didáticos dos capítulos 2 a 9, em sua maioria, foram escritos para utilizar o hardware mínimo proposto nas figuras 1.1 e 1.2.

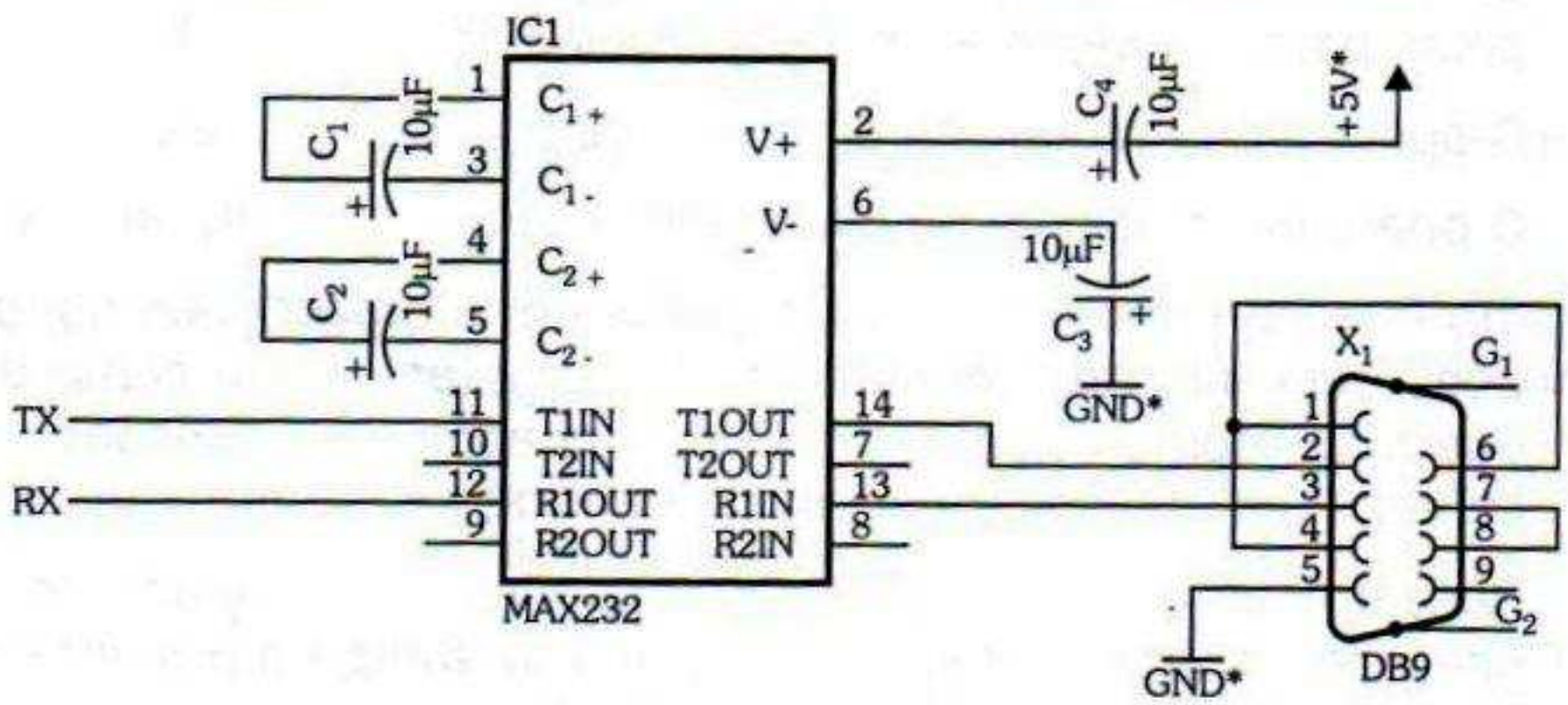

Figura 1.1

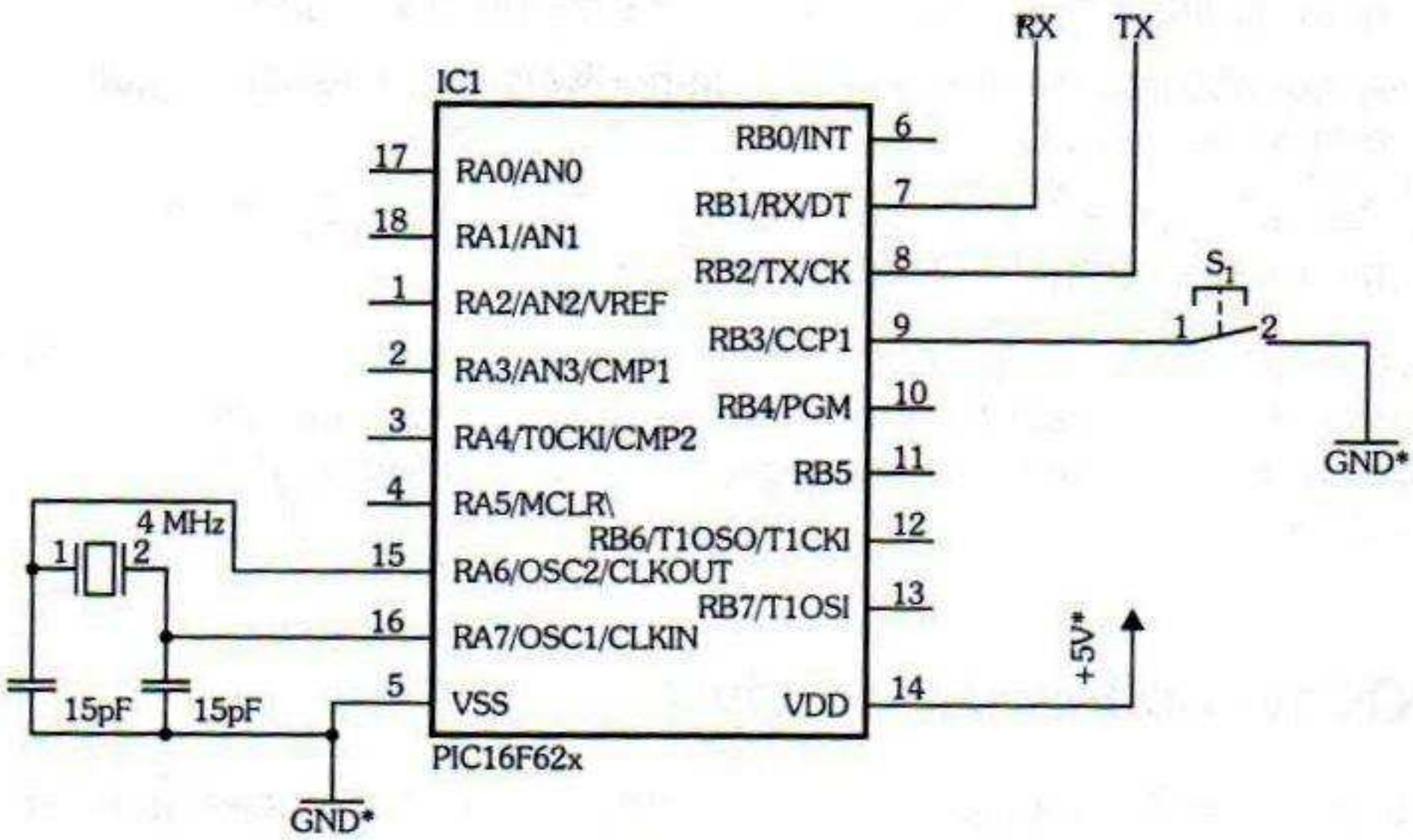

Figura 1.2

É possível utilizar o oscilador interno disponível nos PICs 16F62x, mas devido ao fato de os mesmos não possuírem uma calibração confiável, optamos por utilizar um cristal externo, já que as rotinas de comunicação serial são muito afetadas pela imprecisão do clock.

Alguns dos exemplos foram projetados para utilizar o circuito da figura 1.3:

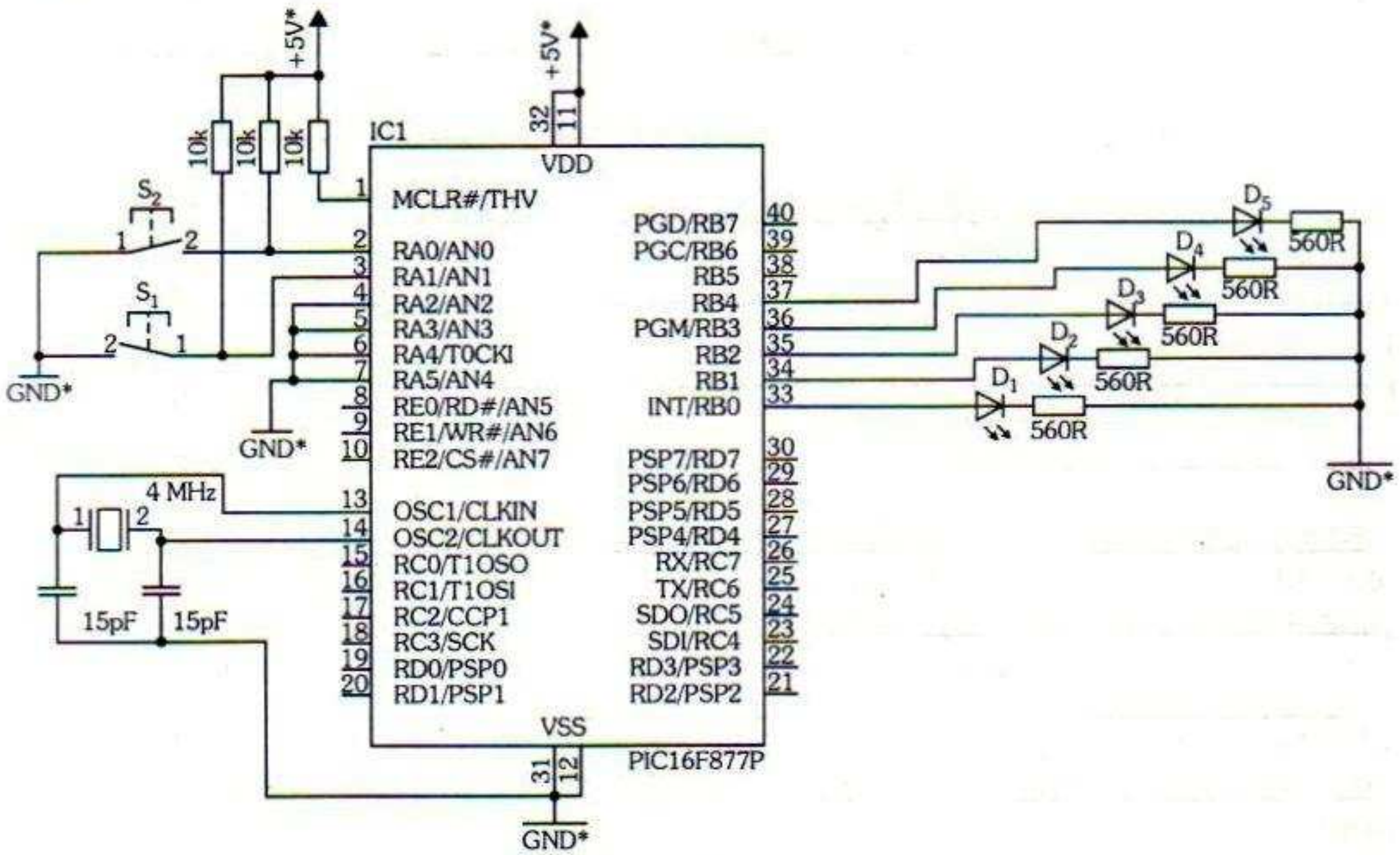

Figura 1.3

Alguns dos exemplos do capítulo 12 podem ser implementados diretamente na placa laboratório PIC Aplicações II da Symphony.

Os softwares programadores foram o IC-PROG e o EPICWIN, utilizados respectivamente com os programadores JDM e PIC Aplicações II.

Anotações

Capítulo 2

Princípios de Programação

Antes de iniciar o estudo da linguagem C propriamente dita, faz-se necessário relembrar os conceitos básicos de programação de sistemas. Neste capítulo vamos realizar um breve estudo sobre temas como álgebra booleana, algoritmos e outros princípios importantes na programação de sistemas.

Podemos dizer que programação, consiste na tradução do pensamento lógico necessário para o cumprimento de determinada tarefa, em uma seqüência de comandos que podem ser interpretados e executados por uma máquina.

Esta afirmação denota, desde já, a idéia de que um programa é fruto do ordenamento lógico e traduzido do raciocínio humano.

De fato, a tarefa de programação constitui-se basicamente num processo de identificação e solução de problemas. Para isso o programador utiliza-se de um conjunto de procedimentos genéricos:

1º) **Exposição do problema**: o programador deve descrever detalhadamente o problema a ser resolvido pelo programa;

2º) **Análise da solução**: sendo conhecido o problema, o programador deve elaborar a ou as soluções que melhor resolvem o problema em questão;

3º) **Codificação da solução**: uma vez que o programador é capaz de descrever tanto o problema quanto a solução para ele, é necessário então a codificação, ou seja, a descrição seqüencial passo a passo da solução que melhor resolve o problema. A isto dá-se o nome de algoritmo;

4º) **Tradução do código**: partindo da seqüência de comandos obtidos do procedimento anterior, resta ao programador traduzir essa seqüência em comandos que possam ser corretamente interpretados pela linguagem de programação utilizada;

5º) **Depuração** (em inglês *DEBUG*): infelizmente, por mais experiente que seja o programador, um programa nunca está livre de erros, por isso, após implementação do programa, seja qual for a linguagem utilizada, será necessário proceder à sua depuração. Depuração então é o processo de verificação e teste do programa de forma a localizar e solucionar todas as eventuais falhas e erros de codificação que tenham acontecido em quaisquer das fases anteriores.

Utilizam-se diversas técnicas para facilitar estas tarefas listadas. Na análise e codificação da solução, por exemplo, utilizam-se diagramas e fluxogramas para a construção gráfica de algoritmos, o que permite uma melhor visualização dos passos envolvidos nele.

Dependendo da complexidade do programa, pode ser necessário utilizar alguns artifícios de programação:

- Um conjunto de comandos utilizado repetidamente, em diferentes partes do programa, pode ser codificado separadamente, formando um módulo ou função. Desta forma, em vez de repetir a seqüência de comandos a cada necessidade, o programador pode simplesmente executar a função especificada;
- Um conjunto de comandos executado diversas vezes numa mesma seqüência constitui, no jargão dos programadores, um laço de repetição ou em inglês *LOOP*. Mais adiante veremos as formas de codificação de um *loop*;
- Normalmente em um programa é necessário verificar ou testar determinada condição, seja comparar dois números, verificar uma senha, etc. A codificação dos testes condicionais, baseados na álgebra booleana, também obedece a algumas regras que veremos adiante.

Durante a fase de codificação do algoritmo, o programador pode utilizar-se de ferramentas, ou programas de computador especiais, denominados geradores de programa, que realizam automaticamente a tarefa de conversão de um algoritmo em códigos ou mesmo comandos de determinada linguagem.

Encontramos também alguns programas especiais e que são utilizados na fase de tradução de código. Estes programas são classificados em duas categorias: interpretadores e compiladores.

As linguagens interpretadas são aquelas em que a tradução da linguagem é feita em tempo real, durante a execução do programa. Isto significa que as linguagens interpretadas são mais lentas, pois o processo de tradução tem de ser feito para cada instrução do programa. Como exemplos de linguagens interpretadas, podemos citar algumas versões da linguagem BASIC, inclusive o BASIC STAMP, disponível para os PICs.

Já as linguagens compiladas são aquelas em que o processo de tradução (compilação) é feito previamente e o código gerado pelo compilador (código de máquina) pode ser carregado na memória e executado diretamente por ele.

Fica clara a vantagem das linguagens compiladas sobre as interpretadas, já que nas interpretadas, o trabalho de tradução do código-fonte, além de ser feito em tempo real, deve ser feito pelo próprio processador que irá executar o código gerado, o que em muitos casos (como no PIC em questão) resulta em uma enorme diferença.

Outro tema interessante sobre as linguagens interpretadas é o aspecto da otimização de código: nas linguagens interpretadas, cada comando de alto nível é traduzido em uma respectiva seqüência de comandos de baixo nível pelo programa interpretador. Se um mesmo comando for utilizado diversas vezes no programa, serão geradas tantas seqüências de comandos de baixo nível quantas forem as aparições do comando de alto nível.

Por outro lado, em um compilador, isto já não acontece da mesma forma. De maneira geral, os compiladores são programas altamente complexos e utilizam diversos artifícios para a otimização do código gerado, tanto em função do espaço ocupado, como também na velocidade de execução. Por isso mesmo, o processo de compilação é complexo demais para ser embutido em um *chip* tão pequeno como os PICs, por exemplo, razão pela qual devemos utilizar plataformas computacionais mais potentes para a execução de tais *softwares* (denominação em inglês para programas de computador).

Finalmente, na fase de depuração, encontramos também diversos programas e *hardware* (equipamentos) para auxiliar o programador na busca de erros e falhas no programa. Alguns desses programas e equipamentos especiais são utilizados para simular a execução do programa, permitindo ao programador a checagem a qualquer tempo do estado do programa, verificação de variáveis, etc.

2.1. Fluxogramas

Como já foi dito, os fluxogramas são ferramentas que auxiliam grandemente a tarefa de codificação de um programa.

Na realidade, fluxogramas são elementos gráficos utilizados para estabelecer a seqüência de operações necessárias para o cumprimento de determinada tarefa e, conseqüentemente, a resolução de um problema.

A seguir, veremos alguns exemplos de elementos utilizados na construção de fluxogramas, bem como alguns exemplos deles.

	Início ou terminação: este tipo de símbolo é utilizado para representar o início ou término do programa ou algoritmo.
	Processo: este símbolo é utilizado para descrever a realização de uma determinada tarefa.
	Dados: normalmente este símbolo é utilizado para descrever um processo de entrada de dados.
	Tomada de decisão: este símbolo é utilizado para representar um ponto de tomada de decisão, ou teste condicional. A tomada de decisão pode conduzir sempre a um resultado: verdadeiro ou falso. Em um dos casos o fluxo do programa é desviado e no outro, normalmente, o programa seguirá na sua seqüência normal.

Vejamos um exemplo de um fluxograma para a solução de um problema simples: somar dois números (A e B) e armazenar o resultado em C.

O fluxograma da figura 2.1 representa os passos necessários para solucionar o problema descrito.

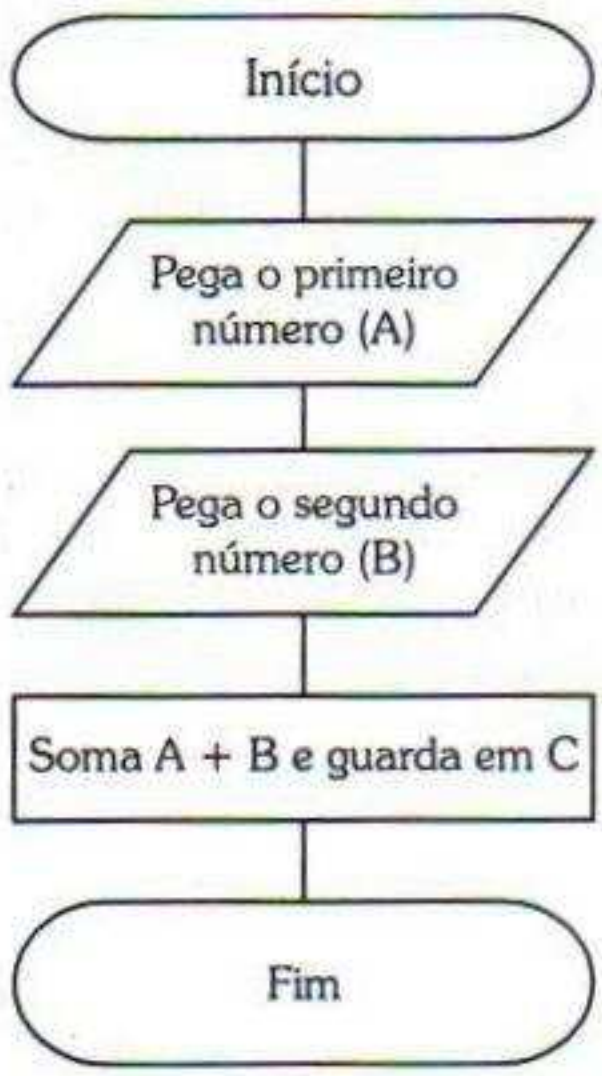

Figura 2.1

Um outro exemplo: um programa para contar de 0 até 10.

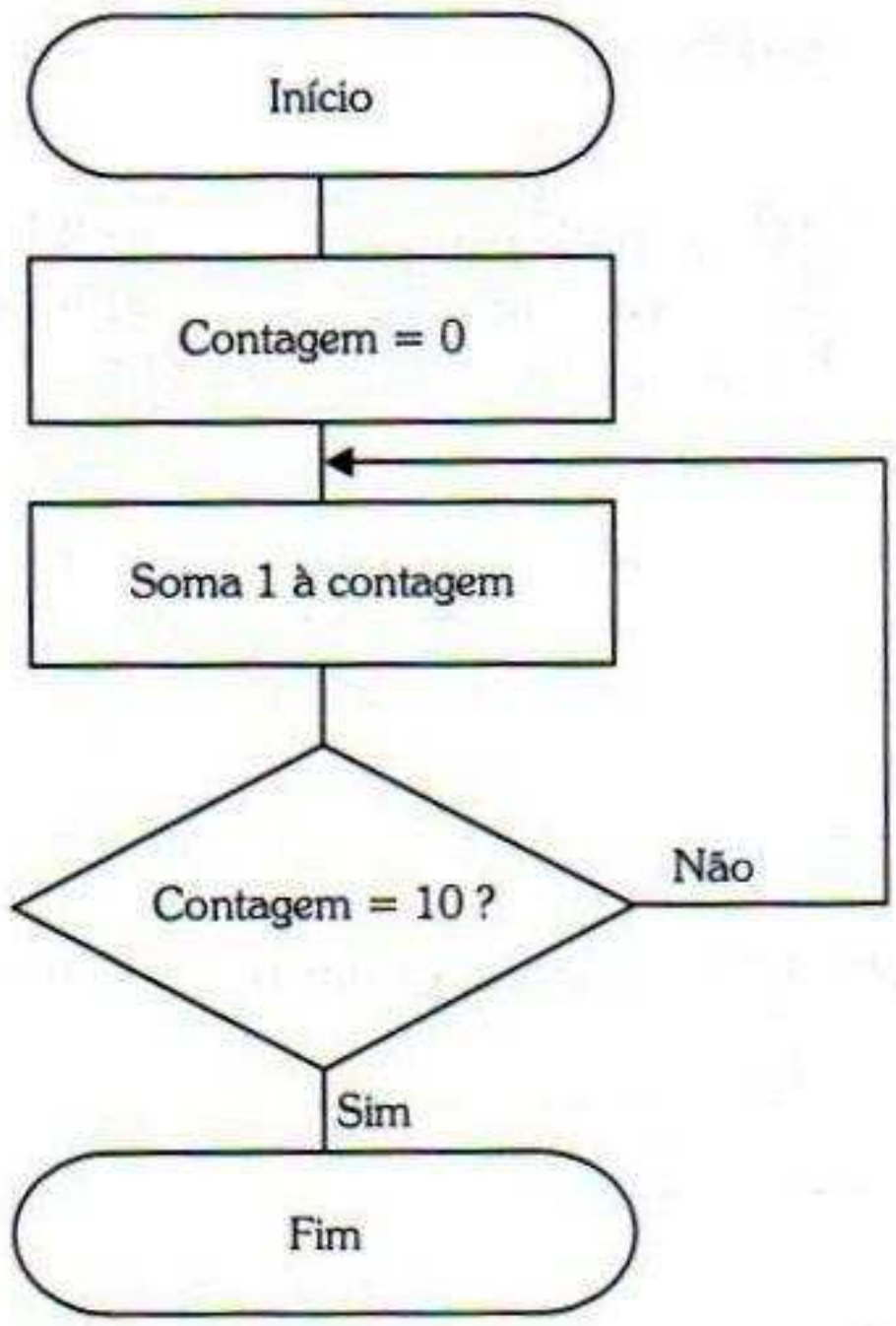

Figura 2.2

2.2. Álgebra Booleana

Agora que já vimos alguns detalhes sobre o funcionamento e aplicação dos fluxogramas no planejamento da solução de um problema, é necessário verificarmos alguns conceitos importantes sobre a resolução de problemas lógicos: a álgebra booleana, um ramo da matemática encarregado da resolução de problemas lógicos. Este nome é devido ao fato de ter sido o inglês George Boole (1815-1864) um dos seus principais articuladores.

A álgebra booleana é baseada em regras e conceitos lógicos simples: proposições e operadores lógicos relacionais.

Uma proposição nada mais é do que qualquer afirmação verbal passível de ser classificada como sendo verdadeira ou falsa. Assim, a afirmação "Hoje é domingo" pode ser considerada uma proposição, já que pode ser classificada como verdadeira ou falsa. Já a frase "Que dia é hoje?" não é uma proposição, pois não pode ser classificada como verdadeira ou falsa.

Duas ou mais proposições podem ser relacionadas com o uso dos operadores lógicos relacionais E, OU e NÃO.

O operador relacional E é utilizado para conectar duas ou mais proposições, resultando na chamada CONJUNÇÃO. A conjunção de duas ou mais pro-

posições somente será verdadeira se TODAS as proposições forem verdadeiras.

Em seguida temos a tabela-verdade, que demonstra o relacionamento entre as proposições "A" e "B", o que resulta na conjunção C. Imagine então que tenhamos a seguinte afirmação: "Se for no domingo E o tempo estiver bom, iremos à praia."

Se considerarmos a afirmação "Se for no domingo" como sendo a proposição "A" e a afirmação "o tempo estiver bom" como sendo a proposição "B", a conjunção C que irá determinar a ida ou não à praia pode ser verificada da seguinte forma:

A (É domingo?)	B (O tempo está bom?)	C (Vamos à praia?)
F	F	F
F	V	F
V	F	F
V	V	V

Tabela 2.1

Já o operador OU é também utilizado para relacionar duas ou mais proposições, resultando em uma nova proposição chamada DISJUNÇÃO. A disjunção de duas ou mais proposições será verdadeira se QUALQUER das proposições for verdadeira.

No exemplo seguinte, temos a utilização da DISJUNÇÃO. Imagine um sistema de alarme composto de dois sensores (A e B). A saída do alarme (C) será verdadeira (alarme disparado) se qualquer dos sensores estiver ativado (verdadeiro).

Vejamos a tabela-verdade para a disjunção:

A	B	C
F	F	F
F	V	V
V	F	V
V	V	V

Tabela 2.2

Podemos verificar que a proposição C (o alarme está disparado) será verdadeira se um dos sensores (A ou B) for verdadeiro.

Finalmente, temos o operador lógico NÃO utilizado para negar uma determinada proposição, fazendo com que a proposição possua sentido contrário ao original. Assim, considerando a proposição original: "Hoje é domingo", se verdadeira, concluiríamos que é domingo e se falsa, que não é. Negando a proposição, temos: "Hoje NÃO é domingo". Observe que agora, se a nova proposição for verdadeira, concluiremos que não é domingo e se falsa, que é domingo, ao contrário portanto, da proposição original.

Em seguida temos a tabela-verdade para a função NÃO:

A	B
F	V
V	F

Tabela 2.3

Observe ainda que a negação de B também pode ser expressa como não B.

Após esta breve explanação sobre alguns dos princípios gerais de programação, vejamos um pouco mais sobre as linguagens de programação.

2.3. Variáveis e Dados

Variável é uma representação simbólica para elementos pertencentes a um determinado conjunto. As variáveis são armazenadas na memória da máquina e podem assumir qualquer valor dentre o conjunto de valores possíveis.

Sempre que necessitarmos armazenar algum tipo de dado, seja ele proveniente do mundo exterior (uma tecla pressionada, uma tensão lida externamente, etc.) como do próprio programa (o resultado de uma equação, por exemplo) será utilizada uma variável.

De fato, as variáveis ficam localizadas na memória do equipamento, não em qualquer tipo de memória, mas nas chamadas memórias RAM. Isto significa que num computador, boa parte da sua memória RAM é ocupada por variáveis, definidas pelos programas em execução no computador.

Um outro aspecto importante sobre as variáveis é que durante a execução de um programa, uma variável pode assumir diversos valores diferentes, mas nunca dois ou mais valores simultaneamente.

As variáveis são classificadas segundo o tipo de conteúdo que armazenam e podem ser: numéricas, caractere, alfanuméricas e lógicas.

As variáveis numéricas são evidentemente utilizadas para o armazenamento de dados numéricos e podem ser classificas em:

- **Inteiras**: quando utilizadas para armazenar valores inteiros. Exemplo: 0, 5, 156, etc.
- **Reais** ou **ponto flutuante**: quando utilizadas para armazenar valores não inteiros, ou seja, números reais. Normalmente, este tipo de variável utiliza uma notação binária especial para o seu armazenamento e utilizam mais posições de memória que as variáveis inteiras.

A quantidade de memória utilizada para armazenar uma variável depende do seu tipo: variáveis inteiras são normalmente encontradas com tamanhos de 8, 16 ou 32 bits. Uma variável inteira de 8 bits pode armazenar um valor entre 0 e 255, uma variável de 16 bits pode armazenar um valor entre 0 e 65535 e uma variável de 32 bits pode armazenar um valor entre 0 e 4.294.967.295.

Observe que os números representados acima são somente positivos, no caso de representação de números com sinal, a magnitude de representação reduz-se à metade daquela sem sinal.

Já as variáveis do tipo real ou de ponto flutuante são normalmente encontradas em tamanhos de 16, 32, 64 ou 80 bits! A título de exemplificação, uma variável de ponto flutuante de 80 bits pode representar valores entre $3{,}4^{-4932}$ e $1{,}1^{4932}$!

Vale lembrar ainda que num sistema de 8 bits, uma variável de 16 bits vai ocupar duas posições de memória e, conseqüentemente, uma variável de 32 bits ocupará quatro posições de memória.

Já as variáveis do tipo caractere são utilizadas para representar um caractere, utilizando normalmente para isso um código especial (normalmente o código ASCII).

As variáveis do tipo caractere podem ser agrupadas em conjuntos de variáveis para formar frases ou textos.

As variáveis do tipo **alfanumérica** são utilizadas para armazenar tanto caracteres como valores numéricos, mas nunca os dois simultaneamente. Sendo assim, uma variável alfanumérica pode num dado momento conter um valor numérico e em outro ponto do programa, ser utilizada para armazenar um caractere.

Finalmente, temos as variáveis do tipo **lógico**, capazes de armazenar um estado lógico: verdadeiro ou falso.

2.4. Operadores

Muitas vezes necessitamos relacionar ou modificar um ou mais elementos como variáveis ou dados em um programa.

Operadores são elementos ou símbolos gráficos utilizados então para relacionar ou modificar um ou mais dados ou variáveis.

Podemos classificar os operadores de uma linguagem em cinco categorias principais:

- **Matemáticos**: são utilizados para efetuar determinada operação matemática em relação a um ou mais dados. Exemplo: adição, subtração, etc.;
- **Relacionais**: são utilizados para relacionar dois elementos ou dados. Exemplo: maior, menor, igual;
- **Lógicos**: são utilizados para efetuar operações lógicas booleanas entre dois ou mais elementos ou dados. Este tipo de operação somente pode chegar a um dos resultados: verdadeiro (1) ou falso (0), e são freqüentemente utilizados na criação de testes condicionais com múltiplas variáveis. Exemplo: função E, função OU, função Não;
- **Lógicos bit a bit**: utilizados para realizar operações lógicas bit a bit entre um ou mais elementos ou dados, conforme o caso. As operações lógicas bit a bit são realizadas entre cada um dos bits dos dados, resultando em um valor qualquer. Exemplo: função E, função OU, função Não, rotação de bit, etc.;
- **Memória**: são operadores utilizados para efetuar operações em relação à memória do equipamento. Exemplo: atribuição (=), & e *.

2.5. Exercícios

1) Nas equações booleanas seguintes, quais valores de A, B e C tornam R verdadeira?

I. R = (A e B e C)
   a) A=V, B=F, C=V
   b) A=F, B=F, C=F
   c) A=F, B=V, C=V
   d) A=V, B=V, C=V

II. R = ((A e (não B)) ou C)
   a) A=F, B=F, C=F
   b) A=F, B=F, C=V
   c) A=F, B=V, C=F
   d) A=V, B=F, C=F

III. R = ((não A) ou (não B) ou (não C))
   a) A=V, B=V, C=V
   b) A=F, B=F, C=F

c) A=V, B=F, C=F
d) A=V, B=V, C=F

IV. R = ((A ou B) e (não C))
a) A=V, B=F, C=V
b) A=F, B=F, C=V
c) A=F, B=V, C=F
d) A=V, B=V, C=F

2) Supondo a afirmação: se não estiver chovendo e eu sair mais cedo do trabalho, iremos à praia. E considerando A= "Está chovendo?", B = "Eu sai mais cedo do trabalho?" e C a resposta: "Iremos à praia", determine a equação booleana que descreve corretamente a afirmação inicial:

a) A e B = C
b) A ou B = C
c) A e (não B) = C
d) (não A) e B = C

3) Qual o tipo de variável necessário para armazenar cada um dos seguintes dados:

a) 150
b) "Linguagem C"
c) 100,1
d) 250
e) "123"
f) V
g) –5
h) F

Capítulo 3

Compiladores CCS C

Conforme já dito, neste livro utilizaremos como plataforma de desenvolvimento o compilador PCWH da CCS. Ele consiste em um ambiente integrado de desenvolvimento (IDE) para o sistema operacional Windows e suporta toda a linha de microcontroladores PIC (séries PIC12, PIC14, PIC16 e PIC18). Existe previsão para suporte aos DsPIC assim que estiverem disponíveis. Os PICs da série 17, além dos microcontroladores da UBICOM/SCENIX (SX), não são suportados por esta versão do compilador.

O IDE na realidade é constituído de três módulos compiladores independentes:

- PCB: para dispositivos de 12 bits (séries PIC12 e PIC16C5X);
- PCM: para dispositivos de 14 bits (séries PIC 14000 e PIC16xXXX);
- PCH: para dispositivos de 16 bits (série PIC18).

O usuário pode optar por adquirir os módulos separados ou o pacote completo, dependendo da aplicação a que se destina o compilador.

Existe também a possibilidade de utilizar o compilador pela linha de comando e também a partir do ambiente MPLAB da Microchip, conforme veremos adiante.

Vejamos agora as principais características do compilador em estudo:

- Compatibilidade com a padronização ANSI e ISO (algumas características do compilador não fazem parte da normatização ANSI devido ao fato de serem específicas para a arquitetura PIC);
- Grande eficiência no código gerado;
- Grande diversidade de funções e bibliotecas da linguagem C (padrão ANSI), tais como: entrada/saída serial, manipulação de strings e caracteres, funções matemáticas C, etc.;

- Grande portabilidade de código entre os diversos microcontroladores PIC e inclusive com código escrito para outros microcontroladores ou sistemas. Isto significa que é muito fácil adaptar um programa escrito em C para outro dispositivo ou sistema (seja ele um outro PIC, outro tipo de microcontrolador, ou mesmo um programa para PC).

3.1. Ambiente Integrado de Desenvolvimento (IDE)

Como já dissemos, o compilador PCWH é constituído de um IDE gráfico que pode ser executado em qualquer plataforma Windows™.

Na figura 3.1 temos uma imagem do IDE.

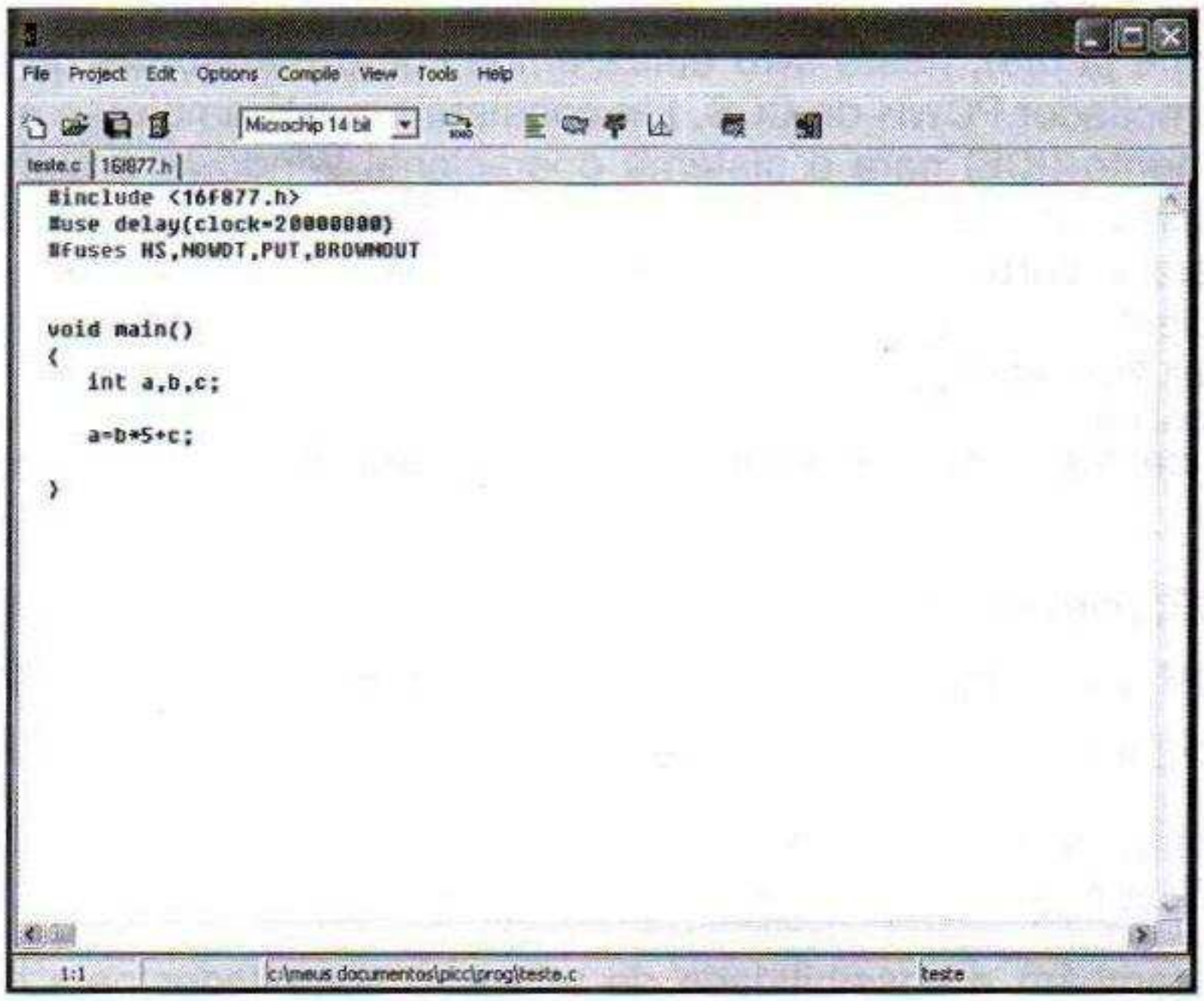

Figura 3.1

Passemos então a uma análise detalhada de cada opção e ícone disponíveis neste ambiente.

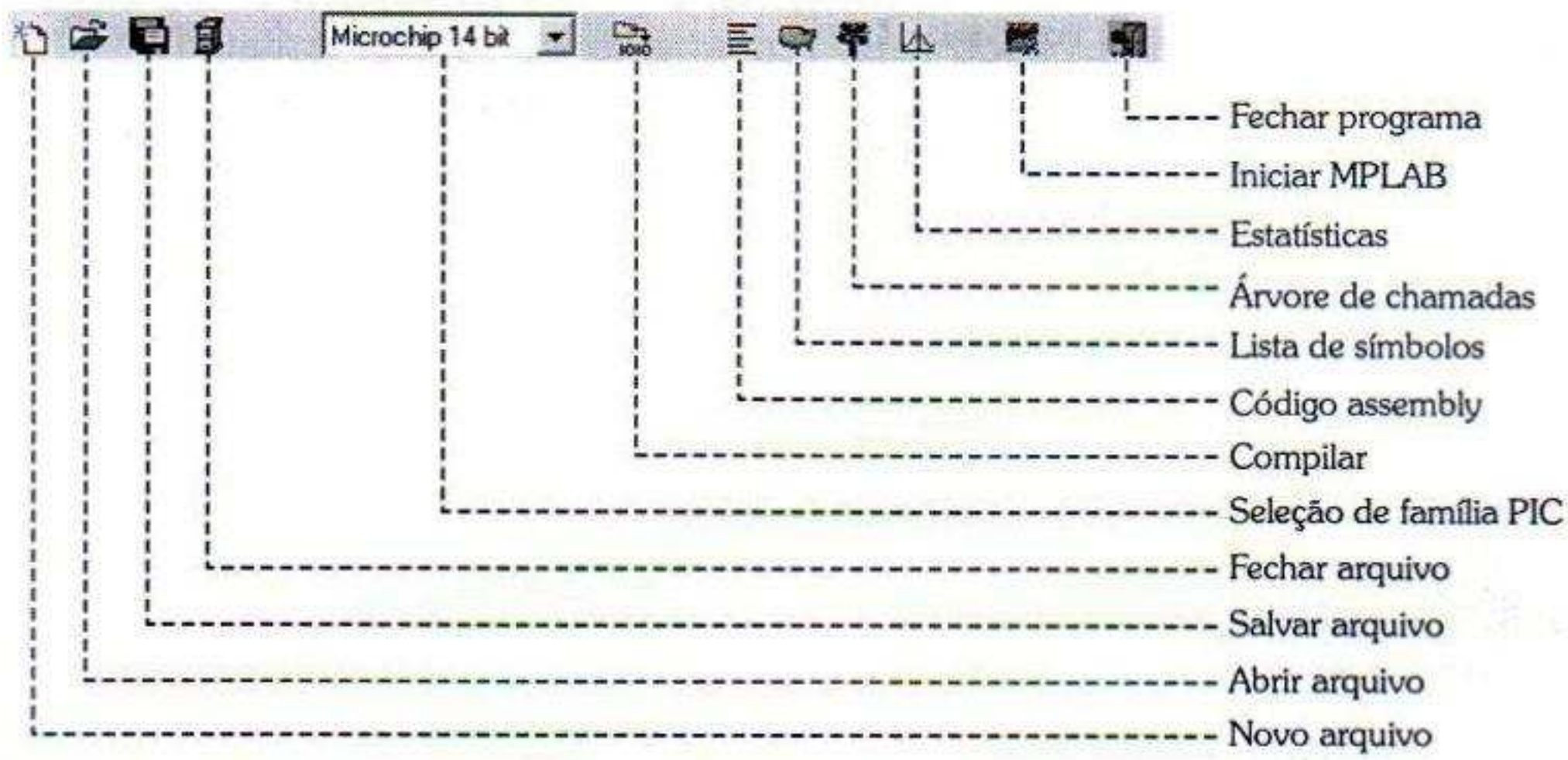

Figura 3.2

A seleção de família PIC é utilizada para especificar qual dos três compiladores (PCB - 12 bits, PCM - 14 bits ou PCH - 16 bits) será utilizado para compilar o programa.

O conjunto de botões é utilizado para visualizar as seguintes informações:

- Visualizar o código *Assembly* gerado pelo compilador. A visualização desse código pode auxiliar o programador na verificação e otimização do código gerado pelo compilador.

No exemplo do programa da figura 3.1, o código *assembly* visualizado é:

```
CCS PCW C Compiler, Version 3.104, 14768

               Filename: c:\meus documentos\picc\prog\teste.LST

               ROM used: 59 (1%)
                         Largest free fragment is 2048
               RAM used: 9 (5%) at main() level
                         10 (6%) worst case
               Stack:    1 locations

*
0000:  MOVLW  00
0001:  MOVWF  PCLATH
0002:  GOTO   main
0003:  NOP
.................... #include <16f877.h>
.................... // Standard Header file for the PIC16F877 device
.................... #device PIC16F877
.................... #list
....................
.................... #use delay(clock=20000000)
.................... #fuses HS,NOWDT,PUT,BROWNOUT
....................
```

```
..................... void main()
..................... {
.....................     int a,b,c;
002B:  CLRF    FSR
002C:  MOVLW   1F
002D:  ANDWF   STATUS,F
002E:  MOVLW   0F
002F:  BSF     STATUS.5
0030:  MOVWF   ADCON1
0031:  BCF     STATUS.5
.....................     a=b*5+c;
0032:  MOVF    b,W
0033:  MOVWF   ??65535
0034:  MOVLW   05
0035:  MOVWF   @MUL88.P1
0036:  GOTO    @MUL88
0037:  MOVF    c,W
0038:  ADDWF   @78,W
0039:  MOVWF   a
..................... }
003A:  SLEEP
```

- Visualizar símbolos gerados: esta opção permite ao programador visualizar a lista de símbolos utilizados e criados pelo compilador para a tarefa de compilação do programa.

No exemplo da figura 3.1, o conjunto de símbolos gerado pelo programa é:

```
008       PSP_DATA
015-016   CCP_1
015       CCP_1_LOW
016       CCP_1_HIGH
01B-01C   CCP_2
01B       CCP_2_LOW
01C       CCP_2_HIGH
021       main.a
022       main.b
023       main.c
024       @MUL88.P1
024       main.@SCRATCH
025       @MUL88.P1
077       @SCRATCH
078       @SCRATCH
078       _RETURN_
079       @SCRATCH
07A       @SCRATCH
07B       @SCRATCH
          delay_ms.P1
          delay_us.P2
002B  main
0004  @MUL88

Project Files:
c:\meus documentos\picc\prog\teste.c
D:\picc\devices\16f877.h
```

- Visualizar árvore de chamadas: esta opção permite ao programador visualizar toda a estrutura de chamada de funções que ocorre dentro do programa, além de apresentar detalhes sobre a utilização de memória RAM e ROM para cada função do programa.

```
└teste
   └main   0/16   Ram=4
      ├??0??
      └@MUL88   0/39   Ram=2
```

Figura 3.3

- Visualizar estatísticas: esta opção permite ao programador visualizar as estatísticas de compilação do programa.

```
ROM used: 59 (1%)
          2048 (25%) including unused fragments
          0 Average locations per line
          59 Average locations per statement
RAM used: 9 (5%) at main() level
          10 (6%) worst case
Lines Stmts  %   Files
----- ----- ---  -----
   13     1 100  c:\meus documentos\picc\prog\teste.c
  249     0   0  D:\picc\devices\16f877.h
----- -----
  524     2 Total

Page ROM  %  RAM  Functions:
---- --- --- ---  ----------
0     16  27   4  main
0     39  66   2  @MUL88


 Segment    Used  Free
---------   ----  ----
0000-0003      4  0
0004-07FF     55  1989
0800-0FFF      0  2048
1000-17FF      0  2048
1800-1FFF      0  2048
```

Vejamos agora a estrutura de menus do compilador PCWH:

File Project Edit Options Compile View Tools Help

Figura 3.4

No menu **FILE** encontramos os tradicionais comandos de criação, abertura, fechamento e impressão de arquivo.

O menu **PROJECT** apresenta comandos para a criação, abertura, fechamento e impressão de arquivos de projeto. No menu PROJECT encontramos

também a opção INCLUDE DIRS, utilizada para especificar em quais diretórios o compilador irá buscar os arquivos de inclusão (normalmente com a extensão ".h").

No meu **EDIT** encontramos os tradicionais comandos de edição (UNDO - desfazer (CTRL+Z), COPY - copiar (CTRL+C), CUT - recortar (SHIFT+DEL), PASTE - colar (SHIFT+INS), SEARCH - procurar (CTRL+F), REPLACE - substituir (CTRL+R), NEXT - procurar próxima (F3)), além de outros, como:

- COPY FROM FILE: copiar de arquivo - insere um arquivo no programa em edição;
- PASTE TO FILE: colar em arquivo - copia o texto selecionado para um determinado arquivo;
- MATCH BRACE (CTRL+]): utilizado para encontrar a abertura/fechamento de chave correspondente à chave atual e mover o cursor para aquela;
- MATCH BRACE EXTENDED (SHIFT+CTRL+]): seleciona todo o bloco correspondente à chave atual;
- INDENT SELECTION (ALT+F8): utilizado para indentar a seleção de texto atual;
- TOGGLE BOOKMARK (CTRL+SHIFT+número): marca um determinado local do programa;
- GOTO BOOKMARK (SHIFT+número): desloca o cursor para um dos locais previamente marcados;
- NEXT WINDOW (CTRL+N): apresenta a próxima janela de edição do compilador;
- PREVIOUS WINDOW (CTRL+P): apresenta a janela de edição anterior.

Figura 3.5

No menu **OPTIONS** encontramos diversas opções de configuração do editor de arquivos e do compilador PCW, tais como:

- REAL TABS: utiliza caracteres de tabulação (TAB) em vez de espaços;
- TAB SIZE: especifica o tamanho de cada tabulação;
- AUTO INDENT: ativa a auto-indentação do editor;
- WORDSTAR KEYS: ativa compatibilidade com seqüências de controle do editor do **wordstar**®;
- EDITOR FONT: seleciona o tipo de caractere utilizado no editor;
- SYNTAX HIGHLIGHTING: realce de sintaxe: exibe os comandos C em cores diferentes;
- AUTO HIGHLIGHT BRACKETS: realça as chaves;
- EDITOR COLORS: permite selecionar as cores para cada elemento da linguagem C (identificadores, comentários, etc.);
- RECALL OPEN FILES: atualiza os arquivos abertos;
- TOOLBAR: permite configurar a barra de ferramentas do editor;
- FILE FORMATS: abre uma janela de configuração dos formatos de arquivos de saída do compilador:

Figura 3.6

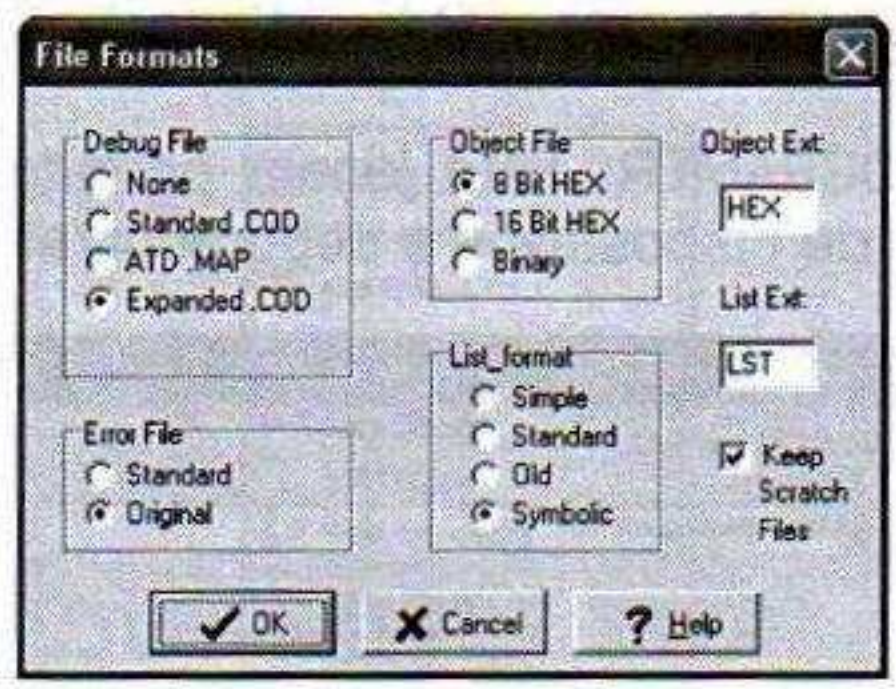

Figura 3.7

Nesta janela podemos definir parâmetros, como: o tipo de arquivo de *debug* gerado (DEBUG FILE), o tipo de arquivo objeto gerado (OBJECT FILE), a extensão do arquivo objeto (normalmente HEX), o tipo de arquivo de erros, formato do arquivo de listagem (LIST FORMAT), extensão do arquivo de listagem (LST).

O arquivo de *debug* é normalmente utilizado pelos emuladores de hardware.

O arquivo de objeto é o programa em código de máquina pronto para ser gravado na memória do PIC.

O arquivo de listagem é utilizado para verificação do código gerado, incluindo o fonte em C e o *assembly* gerado pelo compilador.

- GLOBAL DEFINES: Esta janela permite a criação de definições padrão para utilização no ambiente IDE. Repare que estas definições não são utilizadas pelo compilador de linha de comando ou pelo ambiente integrado no MPLAB. Nestes casos, o programador deve utilizar definições no próprio código-fonte do programa ou em um arquivo separado.

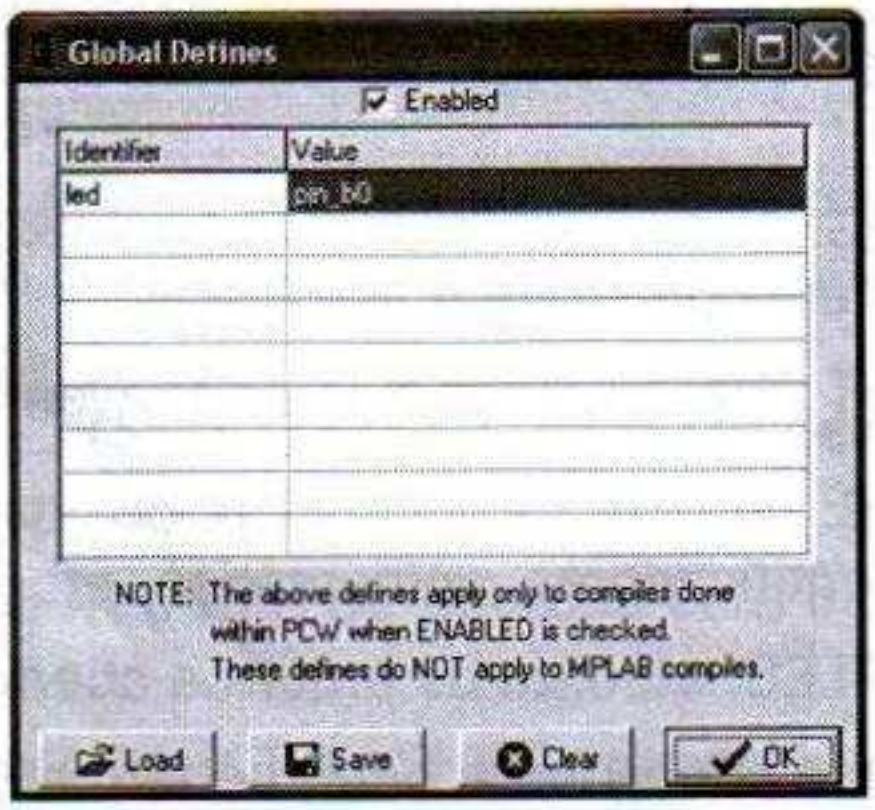

Figura 3.8

Convém observar também que estas definições somente são utilizadas quando a opção ENABLED está selecionada.

- DEBUGGER/PROGRAMER: Esta janela permite selecionar as opções de programador de dispositivos e depurador/simulador.

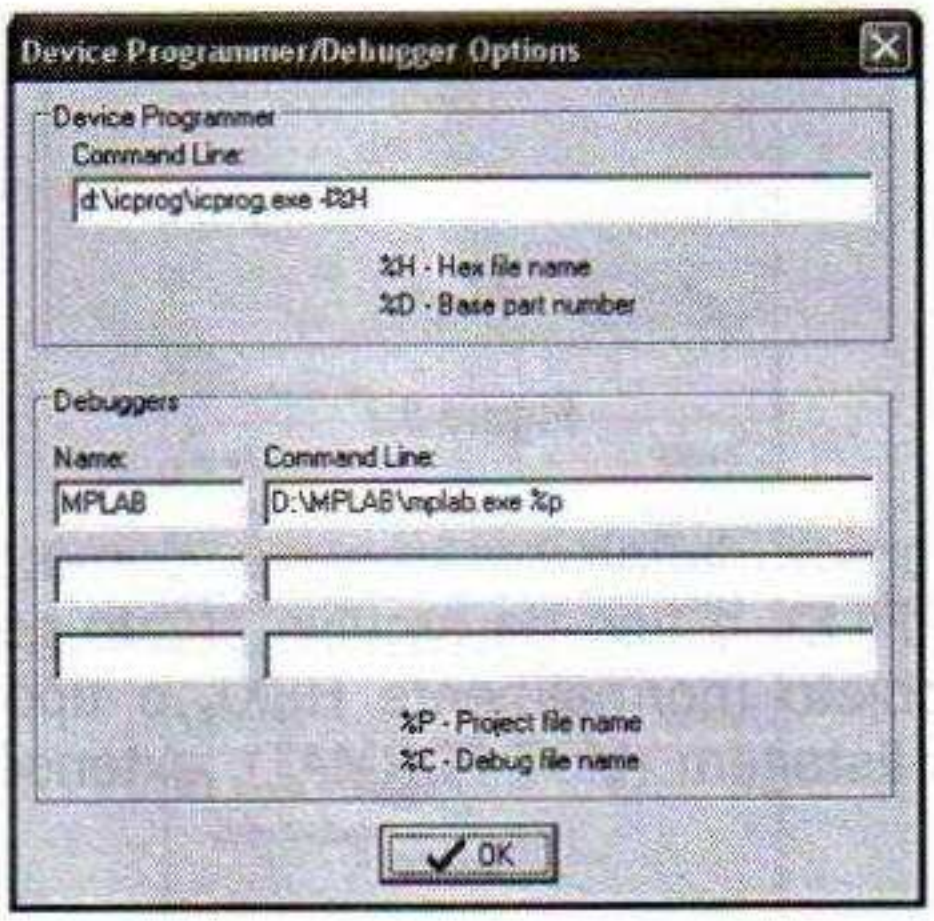

Figura 3.9

No caso do programador, é possível adicionar ainda parâmetros como o nome do arquivo a ser programado (%H) e o dispositivo (%D).

No caso do debugador/simulador, é possível adicionar parâmetros como o nome do projeto (%P) e nome do arquivo de debug (%C).

- INCLUDE DIRS: Esta janela permite alterar e adicionar a ordem e os diretórios de pesquisa para os arquivos de inclusão. A ordem de pesquisa é de cima para baixo.

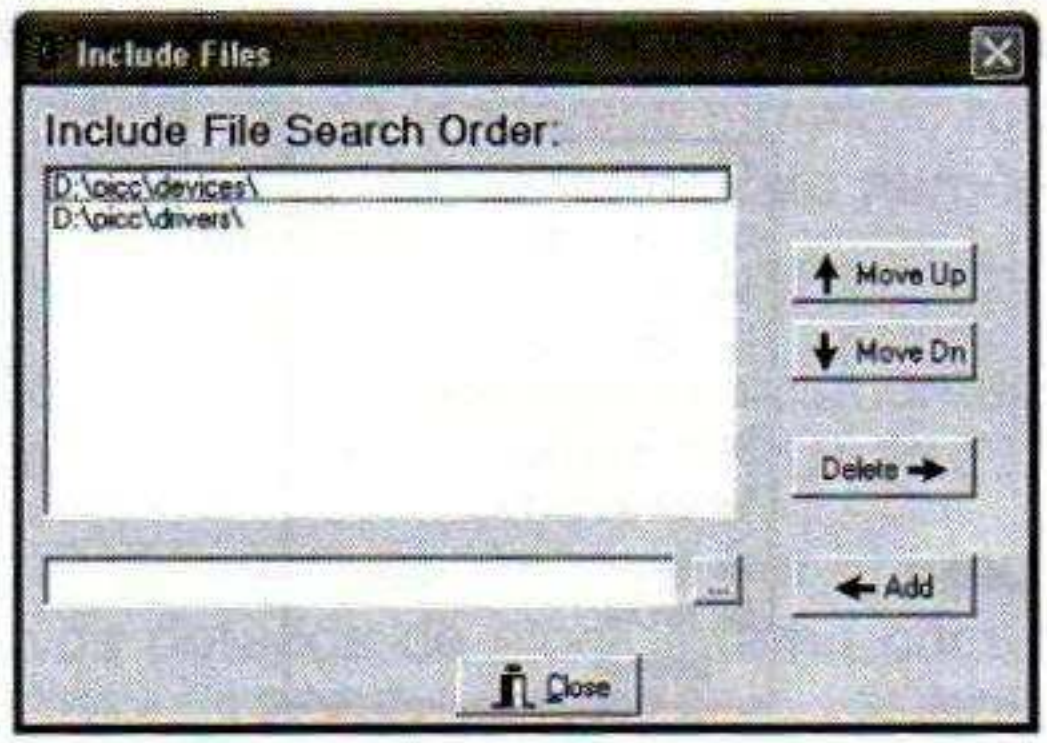

Figura 3.10

A próxima opção do menu principal é o menu **COMPILE**, que conta com apenas uma opção COMPILE. Ela é obviamente utilizada para dar início ao processo de compilação do programa.

A seguir, temos o menu VIEW com as seguintes opções:

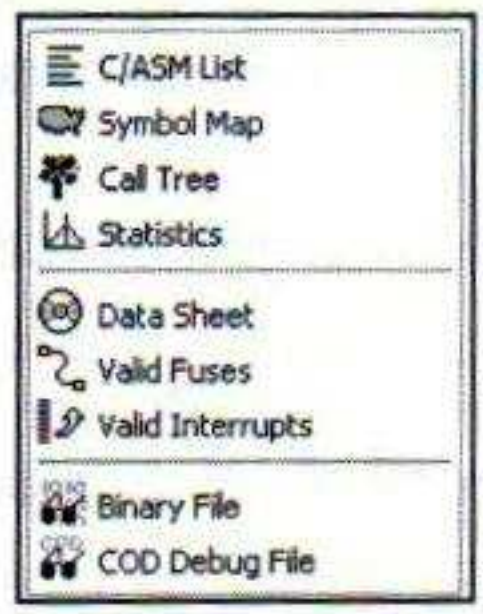

Figura 3.11

As quatro primeiras opções já foram vistas anteriormente neste capítulo.

Neste menu, encontramos ainda as opções:

- DATA SHEET: para visualizar a folha de dados do dispositivo em uso (é necessário instalar separadamente os datasheets dos componentes, o que pode ser feito a partir do CD-ROM do compilador ou baixando cada arquivo do site do fabricante);

- VALID FUSES: esta opção permite visualizar as definições válidas para a palavra de configuração do dispositivo selecionado;
- VALID INTERRUPTS: permite visualizar as interrupções disponíveis no dispositivo selecionado;
- BINARY FILE: abre um arquivo binário para visualização;
- COD DEBUG FILE: abre um arquivo debug para visualização.

No menu **TOOLS** encontramos algumas ferramentas internas e externas muito interessantes e que auxiliam o programador na tarefa de programação:

Figura 3.12

- DEVICE EDITOR: o editor de dispositivos é uma ferramenta muito interessante que permite ao programador adicionar ou alterar a forma como o compilador trata cada dispositivo da família PIC. Com esse editor é possível adicionar novos dispositivos à medida que eles se tornarem disponíveis;
- DEVICE SELECTOR: esta opção abre uma janela que permite ao programador selecionar e comparar as características de cada dispositivo das famílias PIC;
- FILE COMPARE: permite efetuar a comparação entre dois arquivos;
- NUMERIC CONVERTER: esse pequeno programa permite efetuar a conversão entre os três principais tipos de dados disponíveis na linguagem C: inteiros sem sinal, inteiros com sinal e ponto flutuante;
- SERIAL PORT MONITOR: invoca um pequeno e simples terminal de comunicação serial, que pode ser utilizado para efetuar a comunicação com um microcontrolador por meio da porta serial do microcomputador;
- DISASSEMBLER: o disassembler é um programa que faz o contrário do montador, ou seja, ele desmonta os códigos binários e traduz novamente em mnemônicos *assembly*.

Figura 3.13

Para utilizar o disassembler, primeiramente informamos o nome do arquivo a ser desmontado em INPUT FILE, em seguida o nome do arquivo a ser gerado em OUTPUT FILE.

Em seguida, selecionamos as opções de desmontagem: código de 12 ou 14 bits (não é possível desmontar programas para PICs da família 18), tipo de arquivo de entrada (hexadecimal ou binário), tipo de rótulos (LABELS) utilizado (endereço simples, rótulo C ou Assembly) e finalmente o formato numérico de saída.

Ao clicar no botão DASM, o processo de desmontagem terá início e será gerado o arquivo de saída especificado em OUTPUT FILE.

- EXTRACT CAL DATA: esta opção deve ser utilizada quando se trabalha com dispositivos dotados de clock interno com palavra de calibração (12C50x, 12C51x, 12C67x, 12F629 e 12F675, entre outros). Uma vez selecionada esta opção, o programador deve informar o nome do arquivo hexadecimal (que deve conter a leitura prévia do *chip* de destino) do qual será feita a extração da palavra de calibração (normalmente localizada no último endereço válido da memória de programa);
- PROGRAM CHIP: invoca o programador de dispositivos (o qual não faz parte do pacote do compilador e deve ser adquirido separadamente);
- MPLAB: invoca o ambiente MPLAB;
- INTERNET: permite acesso ao site do fabricante do compilador.

Finalmente, a última opção do menu principal do programa, **HELP**, contém o de praxe: auxílio (em inglês) ao programador.

3.2. Integração com o MPLAB

Além do ambiente IDE, é possível utilizar os compiladores PCB, PCM e PCH integrados ao ambiente MPLAB.

A seguir, veremos passo a passo os procedimentos necessários para programar em linguagem C dentro do MPLAB.

Após instalado o compilador no computador, inicie o MPLAB e selecione a opção PROJECT > NEW PROJECT.

Digite o nome do arquivo de projeto e clique em OK. Deve surgir a janela de opções do projeto, como na figura 3.14.

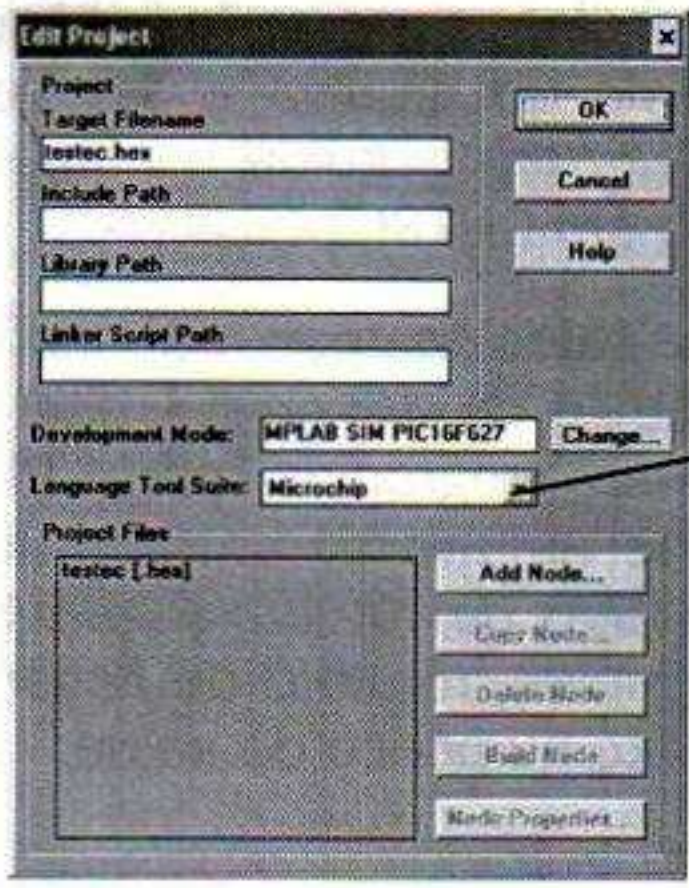

Figura 3.14

Figura 3.15

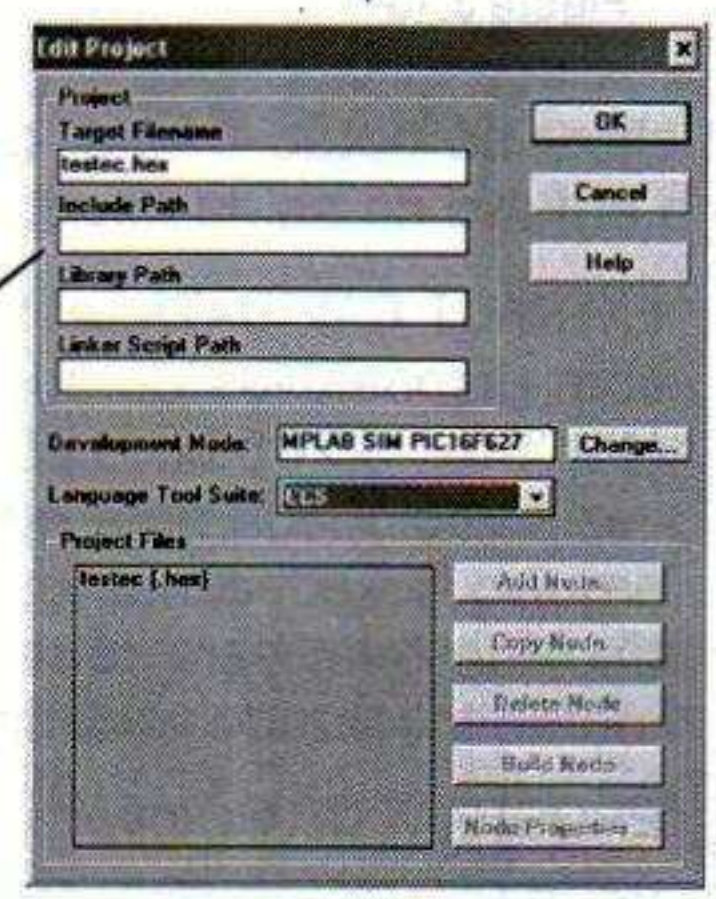

Figura 3.16

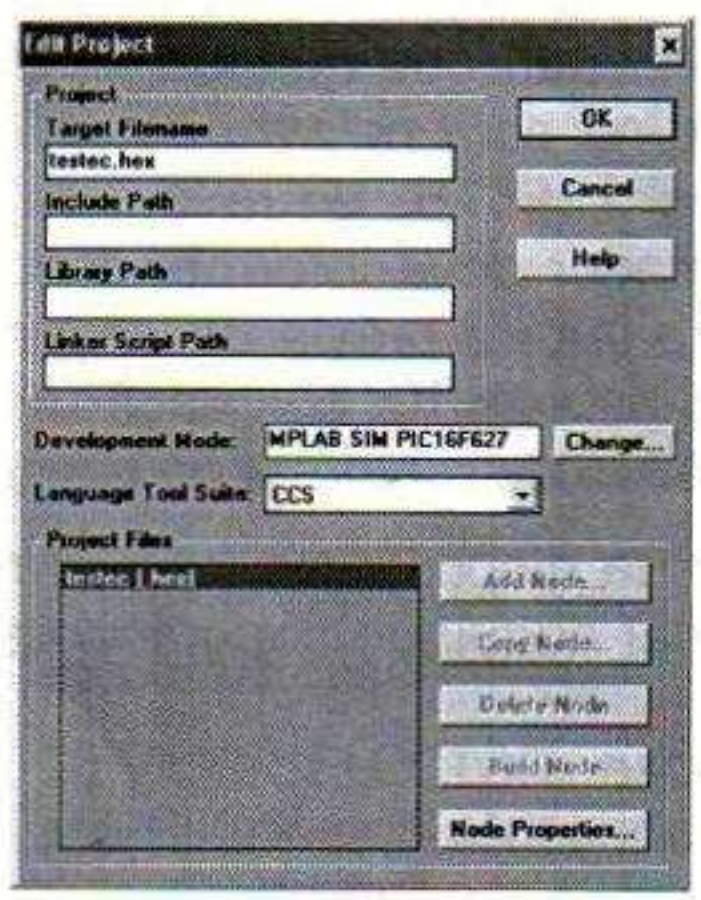

Figura 3.17

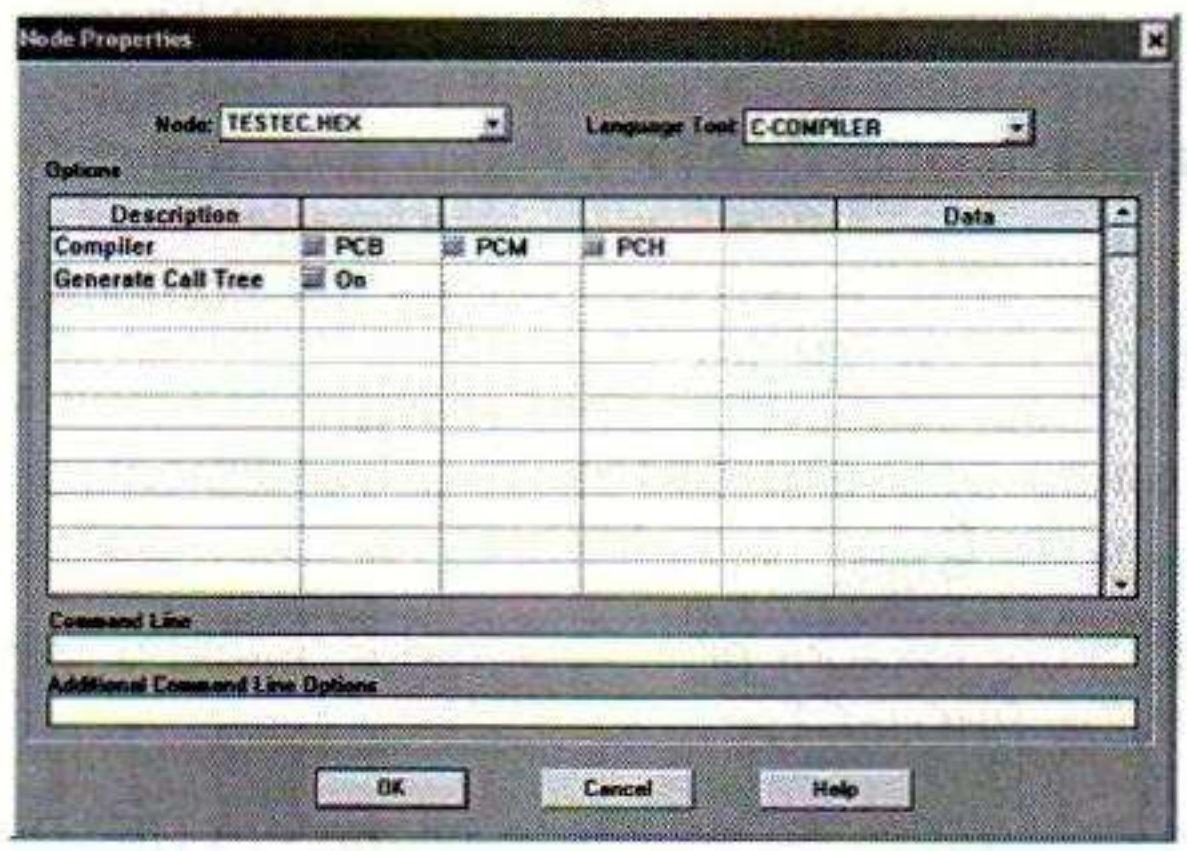

Figura 3.18

Em seguida, clique na opção LANGUAGE SUITE e selecione o compilador CCS (figura 3.16). Surgem mensagens de alerta (normalmente três); clique apenas OK.

Repare que após a modificação da linguagem de programação, a opção ADD NODE, que anteriormente estava ativada, vai agora aparecer desativada como as outras.

Clique no nó principal do projeto (o arquivo testec[.hex]) destacado na figura 3.17 e selecione a opção NODE PROPERTIES.

Deve surgir a janela de propriedades do nó do projeto (figura 3.18). Nessa janela o programador seleciona qual dos compiladores será utilizado (PCB, PCM ou PCH, conforme o pacote adquirido). É possível também acionar a opção GENERATE CALL TREE para fazer com que o compilador gere um arquivo com a árvore de chamadas de funções.

Agora, basta selecionar qual o modo de funcionamento do MPLAB (em DEVELOPMENT MODE) clicando no botão CHANGE e selecionando o dispositivo e o modo desejado.

Se já houver um programa-fonte em C salvo no HD, o programador pode então selecionar a opção ADD NODE e adicionar o arquivo desejado. Se não houver arquivo, clique em OK e salve o projeto (PROJECT > SAVE PROJECT).

Caso não tenha digitado o programa ainda, o programador pode fazê-lo utilizando a opção FILE > NEW, salvando em seguida o arquivo e adicionando-o ao projeto, como mencionado anteriormente.

Anotações

Capítulo 4

Introdução à Linguagem C

Agora que o leitor já possui o conhecimento sobre o funcionamento do compilador e do ambiente de desenvolvimento, é hora de estudar os fundamentos da linguagem C.

Para facilitar a compreensão da linguagem C, vejamos primeiramente um exemplo de programa:

Exemplo 4.1

```
// Primeiro exemplo

#include <16F628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,BROWNOUT,NOLVP,NOMCLR

main()
{
   int tempo;
   tempo=100;      // atribui o valor 100 à variável tempo
   while (true)
   {

         output_high(PIN_B0);  // seta o pino RB0
         delay_ms(tempo);      // aguarda 100ms
         output_low(PIN_B0);   // resseta o pino RB0
         delay_ms(tempo);      // aguarda 100ms
   }
}
```

Vejamos o significado de cada linha do programa.

A primeira linha do programa:

```
// Primeiro exemplo
```

É chamada de comentário. Os comentários são descrições inseridas no código-fonte pelo programador com o intuito de documentar o programa.

Esta versão de compilador C admite duas formas de comentário:

- Comentários de linha simples: são de apenas uma linha iniciados pelos caracteres "//", tais qual o comentário do programa anterior. Este tipo de comentário não faz parte da padronização ANSI original, mas atualmente é encontrado em vários compiladores.

  Os comentários de linha simples podem ser iniciados em qualquer ponto de uma linha e são muito utilizados para descrever o funcionamento ao final de cada linha de código, como podemos observar no decorrer do programa.

- Comentários de múltiplas linhas: são compostos por uma ou mais linhas. Estes tipos de comentário utilizam a seqüência de caracteres "/*" para iniciar o comentário e a seqüência "*/" para terminar o comentário.

Exemplo:

```
/*
   Este é um
   comentário
   de múltiplas
   linhas
*/
```

Na próxima linha temos:

```
#include "16F628.h"
```

O comando **#include** é uma diretiva do compilador. Neste caso está determinando ao compilador que anexe ao programa o arquivo especificado: "16F628.h".

Arquivos com a extensão "h" são chamados de arquivos de cabeçalho e são utilizados em C para definir variáveis, tipos, símbolos e funções úteis ao programa.

O "16F628.h" é um arquivo com as definições relativas ao processador-alvo, para o qual o programa será compilado. Podemos verificar que o processador para o qual o programa foi escrito é o PIC 16F628. Se desejássemos modificar o processador-alvo, bastaria alterar esta linha do programa.

Observação: Ao alterar o processador utilizado no programa, pode ser necessário também, alterar a família do processador na opção disponível na barra de ícones do compilador (figura 3.2).

Na próxima linha encontramos:

```
#use delay(clock=4000000)
```

A seqüência **#use** especifica uma diretiva interna do compilador. No caso, a diretiva determina ao compilador que considere o valor de 4.000.000 de Hertz ou 4MHz para a freqüência de clock do MCU. Este valor é utilizado para geração dos códigos de atraso e outras rotinas que dependam de tempo (como, por exemplo, a rotina de comunicação serial por software).

Em seguida temos:

```
#fuses INTRC_IO,NOWDT,PUT,BROWNOUT,NOLVP,NOMCLR
```

Esta é uma diretiva que especifica o estado dos "fusíveis" da palavra de configuração do dispositivo. No presente caso, estamos selecionando pela ordem: oscilador de clock interno de 4MHz com pinos RA6 e RA7 disponíveis para E/S, watchdog desligado, timer power-up ligado, reset por brown-out ligado, programação por baixa tensão desligada e MCLR interno (pino RA5 disponível como entrada). Estas opções variam conforme o modelo de PIC utilizado.

A lista com os valores válidos para a determinação do estado de cada fusível pode ser verificada a qualquer momento no menu VIEW > Valid Fuses.

Na próxima linha do programa encontramos:

```
main()
```

A declaração **main()** especifica o nome de uma função. No caso, a função **main()** é padronizada na linguagem C e é utilizada para definir a função principal, ou o corpo principal do programa.

Uma função, em C, é um conjunto de instruções que pode ser executado a partir de qualquer ponto do programa.

O sinal de abertura de chave "{" é utilizado para delimitar o início da função e o sinal de fechamento de chave "}" indica o final da função.

Na realidade, as chaves delimitam o que chamamos de bloco de programa, ou bloco de código.

Nas linguagens de programação estruturada (como C), um bloco de programa consiste em um conjunto de comandos conectados logicamente e que assumem uma identidade única, ou unidade. Assim, para todos os efeitos, um bloco de comandos e um único comando funcionam da mesma forma.

Podemos dizer que um programa em C é constituído por um ou mais dos seguintes elementos:

- Operadores:

  Operadores são elementos utilizados para comandar interações entre variáveis e dados em C.

  Muitos dizem que um dos elementos que mais se destacam na linguagem C é a força e flexibilidade dos seus operadores.

  O capítulo 6 faz um estudo sobre os operadores de dados encontrados na linguagem C.

- Variáveis:

  É claro que não podemos escrever um programa realmente útil sem utilizar variáveis para o armazenamento de dados.

  A linguagem C dispõe de uma grande variedade de tipos de variáveis e dados, o que permite o desenvolvimento de praticamente qualquer tipo de aplicação.

  No próximo capítulo, faremos um estudo detalhado das variáveis e tipos de dados encontrados na linguagem C.

- Comandos de controle:

  Os comandos ou declarações de controle são elementos essenciais à escrita de programas em C.

  Como seu próprio nome diz, eles são utilizados para controlar, testar e manipular dados e informações dentro do programa.

  No capítulo 7 estudaremos os diversos comandos de controle disponíveis na linguagem C.

- Funções:

  As funções, como já foi dito, são estruturas de programa utilizadas para simplificar, otimizar ou apenas tornar mais claro o funcionamento do programa.

  No capítulo 9 faremos um estudo mais profundo sobre o funcionamento e utilidade das funções na linguagem C.

A primeira linha do bloco de comandos da função **main()** é:

```
while (true)
```

Este é um comando de controle utilizado na repetição de um determinado bloco de instruções. Esse bloco será repetido enquanto a avaliação da condição especificada entre parênteses for verdadeira. No caso, a avaliação é explicitamente verdadeira (true).

O bloco de instruções que será repetido é aquele especificado dentro das chaves que seguem o comando **while**.

Por ora, isto é tudo o que devemos saber sobre o comando **while**. Quando estudarmos as declarações de controle, retornaremos ao assunto.

Como já dissemos, os comandos que formam o bloco de instruções a serem repetidas são aqueles entre as chaves:

```
{
   output_high(PIN_B0);
   delay_ms(tempo);
   output_low(PIN_B0);
   delay_ms(tempo);
}
```

E serão repetidos indefinidamente até que o processador seja desligado ou ressetado.

Um detalhe interessante a ser observado é que ao final de cada uma das três chamadas de função, o último caractere da linha foi um ponto-e-vírgula (;). Em C o ponto-e-vírgula é utilizado para delimitar o final de um comando.

Observe que as diretivas do compilador não necessitam (nem permitem) do uso do ponto e vírgula como delimitador.

Analisemos então os quatro comandos que compõem o bloco do comando while:

O primeiro comando, `int tempo`, é chamado de declaração de variável. Este comando determina que o compilador crie uma variável do tipo inteiro **int** chamada `tempo`. Esta operação na realidade irá reservar um dos registradores GPR disponíveis para o armazenamento do valor relativo à variável. O tipo inteiro **int** especifica um tipo de dado de 8 bits com valores compreendidos entre 0 e 255 decimal. Mais adiante neste livro, estudaremos os tipos de dados disponíveis na linguagem C e no compilador CCS.

O nome dado à variável é **identificador** e pode ser composto de letras e números. Mais adiante veremos mais detalhes sobre os identificadores válidos na linguagem C.

Em seguida, temos a linha `tempo=100` que constitui-se em uma operação de atribuição. As atribuições em C são executadas pelo operador de igualdade "=". Desta forma, a linha `tempo=100` fará com que o compilador gere uma seqüência de instruções para fazer com que o valor 100 decimal seja armazenado na variável "tempo".

O primeiro comando, `output_high(PIN_B0)`, é uma chamada a uma função interna do compilador.

Esta função é utilizada para setar (ou seja, colocar em nível lógico '1') um pino do microcontrolador. Isto significa que o pino RB0 (da porta B) será setado.

Note que "PIN_B0" é um símbolo predefinido para especificar o pino RB0. Este símbolo está localizado no arquivo de cabeçalho do processador utilizado (no presente caso o arquivo "16F628.h").

Uma característica interessante do CCS C é que o próprio compilador encarrega-se de configurar o pino para funcionar como uma entrada ou saída, dependendo do modo de tratamento de I/O em que se esteja trabalhando.

Outro aspecto interessante desta chamada de função é que ela demonstra uma chamada de função com um parâmetro, que no caso especifica o pino do microcontrolador a ser setado.

Este tipo de parâmetro de uma função é chamado de parâmetro formal. Veremos mais detalhes sobre funções e parâmetros no capítulo 9 deste livro.

A próxima função a ser executada é:

```
delay_ms(tempo);
```

Esta também é uma função interna do compilador e é utilizada para gerar um atraso de X milissegundos. No caso, o atraso será igual a 100ms (já que a variável tempo, utilizada como parâmetro, contém o valor 100).

O próximo comando do bloco é:

```
output_low(PIN_B0);
```

Esta também é uma função interna do compilador e é utilizada para ressetar (ou seja, colocar em nível lógico '0') um pino qualquer do microcontrolador. Novamente neste caso, o pino em questão é o RB0.

Finalmente, o último comando:

```
delay_ms(tempo);
```

Fará com que novamente o compilador gere um atraso de 100 ms.

Maiores detalhes sobre o funcionamento das funções internas do compilador podem ser vistos no capítulo 12.

Desta forma, ao programarmos o PIC com o programa anterior e se tivermos conectado um LED (com o devido resistor de limitação de corrente) ao pino RB0, nós o veremos acender e apagar em uma freqüência de 5 Hz.

4.1. Palavras Reservadas da Linguagem

Toda linguagem de programação possui um conjunto de palavras ou comandos para os quais já existe interpretação interna prévia. Tais palavras não podem ser utilizadas para outras finalidades que não as definidas pela linguagem.

A linguagem C ANSI estipula as seguintes palavras reservadas:

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

Tabela 4.1

4.2. Identificadores

Identificadores são nomes dados pelo programador a variáveis, funções e outros elementos da linguagem C.

Conforme já vimos, um identificador na linguagem C pode ser composto de caracteres numéricos e alfanuméricos. Além de números e letras, o único outro caractere que pode ser utilizado em identificadores é sublinhado "_".

Além disso, um identificador somente pode ser iniciado por uma letra ou sublinhado, nunca por um número. Veja em seguida alguns exemplos de identificadores válidos e inválidos em C:

variavel	válido
variavel1	válido
1abc	inválido
variável	inválido
_teste_2	válido
return	inválido
_123_abc	válido
erro\2	inválido

Tabela 4.2

Nesta tabela o identificador "1abc" é inválido porque se iniciou por um número, "variável" é inválido porque utiliza acento, "return" é inválido porque é uma palavra reservada da linguagem e "erro\2" é inválido porque utiliza o caractere "\".

Repare que não é permitido utilizar como identificador uma palavra reservada da linguagem ou um nome de função já definida em C.

O padrão ANSI determina ainda que apenas os primeiros 31 caracteres do nome do identificador serão utilizados para a diferenciação entre eles.

Isto significa que é possível utilizar identificadores de qualquer tamanho, desde que existam diferenças nos primeiros 31 caracteres, de forma a distinguir entre um e outro.

Vejamos então as principais regras de nomeação de identificadores:

- Só podem ser utilizadas letras, números ou o caractere sublinhado "_";
- Devem sempre ser iniciados por uma letra ou sublinhado;
- Identificadores são únicos; um mesmo identificador não pode ser atribuído a coisas diferentes;
- Não podem ser utilizadas como identificadores as palavras reservadas da linguagem e nomes de funções já definidas;
- Somente os primeiros 31 caracteres do identificador são válidos, no entanto um identificador pode utilizar mais do que este número de caracteres.

4.3. Comandos e Funções Importantes

Antes de prosseguirmos no estudo detalhado da linguagem, é interessante observarmos alguns comandos e funções muito utilizadas nos programas na linguagem C. Estas funções e comandos serão utilizados no decorrer deste livro, daí a importância de termos ao menos uma visão superficial de cada um.

4.3.1. Comando If

O comando **if** (em português "se") é utilizado em testes condicionais do tipo:

```
if (condição) comando;
```

O que significa: se a condição for verdadeira, então execute o comando especificado.

Esta condição deve ser qualquer expressão que possa ser avaliada como verdadeira ou falsa, ou seja, uma proposição booleana.

No capítulo 7 veremos um estudo detalhado do comando **if**.

4.3.2. Comando While

O comando **while** (em português "enquanto"), conforme já visto anteriormente, é utilizado para especificar uma estrutura ou laço de repetição, em que um ou mais comandos são repetidamente executados **enquanto** uma determinada condição for avaliada como verdadeira. A sua forma geral é:

```
while (condição) comando;
```

O que significa: enquanto a condição for verdadeira, execute o comando especificado.

Esta condição é novamente qualquer expressão que possa ser avaliada como verdadeira ou falsa, ou seja, uma proposição booleana.

O comando **while** será melhor estudado no capítulo 7.

4.3.3. Função Printf

A função **printf** é utilizada para possibilitar a saída de dados na linguagem C. A saída de dados é direcionada para o dispositivo padrão de saída, que nos computadores é normalmente o monitor de vídeo. No caso dos PICs, o dispositivo de saída eventualmente disponível é a saída serial.

Assim, a função **printf** é uma excelente forma de transmitir dados serialmente do PIC para um outro dispositivo externo, como, por exemplo, um terminal de vídeo ou um microcomputador.

O formato geral da função **printf** é:

```
printf ( argumento(s) );
```

Em que argumento pode ser uma constante de caracteres direta ou ainda um conjunto de caracteres de controle seguido de uma lista de variáveis.

Veja os exemplos:

```
printf ("Teste"); // imprime a palavra Teste através do dispositivo
                  // de saída padrão
```

Ou,

```
int var_teste;    // declara a variável inteira var_teste
var_teste = 5;    // atribui o valor 5 à variável var_teste
printf ("%d", var_teste);        // imprime o valor da variável var_teste
```

Observe que no último exemplo, a função **printf** utilizou um código de formatação para especificar que o conteúdo da variável deveria ser impresso no formato inteiro. A tabela seguinte apresenta alguns dos códigos de formatação mais utilizados:

Código	Formato de saída
%u	unsigned int
%s	char ou string
%c	caractere (char)
%e	float exponencial

Tabela 4.3

A função **printf** e os seus códigos de formatação serão melhor estudados no capítulo 11.

4.3.4. Função GETC

A função GETC é utilizada para entrada de dados, o que, no caso dos PICs, é feito por comunicação serial.

Esta função aguarda que seja recebido um caractere pela linha serial previamente configurada, retornando o valor do caractere recebido. Veja o formato geral desta função:

```
variável = getc();
```

ou simplesmente,

```
getc();
```

no caso de não ser desejado atribuir o valor retornado pela função (que é o caractere recebido serialmente) a qualquer variável.

Desde já é interessante sabermos que uma função, na linguagem C, pode retornar valores, isto é, após a execução da função, ela pode retornar um valor ou dado referente ao processo realizado.

Por ora, isso é tudo o que você precisa saber sobre a função **getc**. No capítulo 11 veremos detalhadamente o funcionamento desta e de outras funções disponíveis na linguagem C.

Capítulo 5

Variáveis e Tipos de Dados

A linguagem C disponibiliza ao programador uma gama de tipos de dados. A implementação C da CCS possibilita o uso de quase todos os tipos de dados disponíveis em C padrão ANSI.

De fato os tipos de dados disponíveis na implementação C da CCS são os maiores pontos positivos, permitindo a construção de programas de grande complexidade com uma relativa facilidade.

Vejamos os tipos básicos de dados disponíveis:

Tipo	Tamanho em Bits	Intervalo
char	8	0 a 255
int	8	0 a 255
float	32	3.4E-38 a 3.4E+38
void	0	nenhum valor

Tabela 5.1

Os tipos de dados são identificados pelas palavras reservadas da linguagem: **char**, **int**, **float** e **void**.

O tipo **char** é utilizado para representação de caracteres ASCII de 8 bits. Cada variável do tipo **char** pode representar um caractere ASCII. O conjunto de caracteres ASCII é normatizado universalmente e pode ser encontrado no apêndice A do livro.

O tipo **int** é utilizado para representar números inteiros de 8 bits (0 a 255). Estes tipos de dados são amplamente usados em programação C.

Note que o tipo **int** na linguagem C é definido sempre para possuir o tamanho mais eficiente para a arquitetura-alvo: 8 bits no caso dos PICs, 32 bits no caso dos computadores do tipo PC atuais. No entanto, o padrão ANSI especifica que o tipo **int** deve representar um número de 16 bits.

Repare ainda que tanto **char** como **int** representam números inteiros e não podem ser utilizados para representar números fracionários. Para isso, deve ser utilizado o tipo **float**, também chamado de ponto flutuante. Este tipo de dado pode ser utilizado para representar grandezas compreendidas entre $3{,}4^{-38}$ e $3{,}4^{+38}$, tanto para números fracionários como inteiros. Em matemática são os chamados números reais.

Note que este tipo de dado ocupa quatro posições de memória RAM do PIC e além disso, qualquer operação envolvendo variáveis **float** é traduzida pelo compilador em um conjunto complexo de operações.

Sendo assim, o uso de variáveis **float** deve ser evitado ao máximo e restrito apenas às operações que realmente necessitarem de um tipo de dados como este.

Temos ainda o tipo **void**, utilizado normalmente em funções para declarar que ela não deve retornar nenhum valor. Maiores detalhes sobre a sua função serão vistos mais adiante neste livro.

Observação: A linguagem C prevê ainda o tipo double, que consiste em uma versão com capacidade de representação dobrada (64 bits) em relação ao tipo float. O compilador CCS não suporta este tipo de dado, no entanto, a palavra double continua entre as palavras reservadas da linguagem.

5.1. Modificadores de Tipo

Além dos tipos de dados vistos na tabela 5.1, podemos utilizar comandos especiais da linguagem C para modificar os tipos básicos, de forma a obtermos outros tipos de dados.

Esses comandos especiais são chamados de modificadores e são os seguintes: **signed**, **unsigned**, **short**, e **long**.

O modificador **signed** pode ser utilizado para modificar um tipo base de dados de forma que ele possa representar tanto números positivos como números negativos.

A representação de números negativos é feita tomando o bit MSB (***M**ost **S**ignificant **B**it* ou Bit mais significativo) da variável para representar o sinal: bit MSB = 1, sinal negativo, bit MSB = 0, sinal positivo. Nesta notação, chamada complemento de dois, o valor a ser representado é complementado (tem seus bits invertidos um a um) e então é somado 1 ao resultado.

Note que devido ao fato de utilizar um bit para representação do sinal, a magnitude absoluta de representação do tipo modificado será metade da magnitude do tipo não modificado.

Assim, um tipo de dados **signed int** pode representar valores entre -128 e +127, em vez de 0 a 255.

O modificador **unsigned** define um tipo de dado sem sinal, o que é o padrão do compilador CCS. Desta forma, não é necessário utilizar o modificador **unsigned**, já que ele não produz qualquer efeito. Note que o padrão ANSI especifica que os tipos de dados padrão da linguagem C são do tipo **signed**.

Já o modificador **short** é utilizado para definir uma variável com tamanho menor que o tipo modificado, ou seja, uma versão reduzida do tipo especificado. Assim, se especificarmos uma variável como sendo do tipo **short int**, ela será uma versão reduzida do tipo **int**, o que no caso do compilador CCS cria uma variável de apenas um bit de tamanho (o que é também chamado de flag ou sinalizador).

Note que esta aproximação difere do C padrão ANSI, já que o foco principal do compilador CCS é a eficiência e as variáveis de 1 bit são, sem dúvida, bem-vindas em um ambiente tão restrito como em um microcontrolador.

Finalmente, temos o modificador **long**, utilizado para ampliar a magnitude de representação do tipo especificado.

Desta forma, um tipo de dados **long int** terá um tamanho de 16 bits, ou seja, irá ocupar duas posições de memória RAM do PIC e terá uma magnitude de representação de 65536 elementos.

Observe ainda que a linguagem C permite a adoção da forma de declaração simplificada; basta que o programador informe o modificador. Neste caso, o compilador assume que o tipo é o **int**. Assim, a declaração **short int** produz o mesmo efeito que **short**, **long int** produz o mesmo efeito que **long** e assim por diante.

5.2. Outros Tipos Específicos do Compilador CCS C

Além dos tipos de dados descritos anteriormente, encontramos no compilador CCS outros tipos de dados, criados especificamente para a eficiência e compatibilidade do compilador junto a dispositivos como os PICs. São eles:

- **int1**: especifica valores de 1 bit (eqüivale ao tipo short int padrão);
- **boolean**: especifica valores booleanos de bit (equivale ao short int e int1);
- **int8**: especifica valores de 8 bits (eqüivale ao tipo int padrão);

- **byte**: especifica valores de 8 bits (eqüivale ao tipo int e int8);
- **int16**: especifica valores de 16 bits (eqüivale ao tipo long int padrão);
- **int32**: especifica valores de 32 bits.

Note ainda que o compilador possui uma diretiva interna "**#type**" que pode ser utilizada para alterar o tamanho dos tipos padrão. Esta diretiva será melhor estudada no capítulo 10.

A seguir, temos uma tabela com todos os tipos de dados disponíveis por padrão no compilador CCS:

Tipo	Tamanho em Bits	Faixa de valores
short int, int1, boolean	1	0 ou 1
char	8	0 a 255
signed char	8	-128 a 127
unsigned char	8	0 a 255
int, int8, byte	8	0 a 255
signed int, signed byte	8	-128 a 127
unsigned int, unsigned	8	0 a 255
long int, int16	16	0 a 65.535
signed long int	16	-32.768 a 32.767
unsigned long int	16	-32.768 a 32.767
int32	32	0 a 4.294.967.295
signed int32	32	-2.147.483.648 a 2.147.483.647
unsigned int32	32	0 a 4.294.967.295
float	32	3.4^{-38} a 3.4^{+38}

Tabela 5.2

Observação:

- Os tipos **boolean** e **byte** são definidos nos arquivos de cabeçalho de cada PIC e são baseados nos tipos **short int** e **int** respectivamente.
- As variáveis inteiras são armazenadas pelo compilador, iniciando pelo LSB no endereço mais baixo de memória e com o MSB no último endereço ocupado pela variável!

5.3. Declaração de Variáveis

Em C, ao contrário de outras linguagens como BASIC, é necessário declarar a variável antes de poder usá-la.

Declarar uma variável nada mais é do que informar ao compilador que uma variável chamada "x" é do tipo "y". O nome da variável identificador, como já foi visto, pode ter até 31 caracteres (letras, números e o caractere "_") e não pode iniciar por um número. A declaração de variáveis em C obedece à seguinte forma:

```
TIPO nome da variável{,outras variáveis};
```

Podemos ainda declarar e inicializar o conteúdo de uma ou mais variáveis:

```
TIPO nome da variável = valor da variável{, outras variáveis};
```

Vejamos então como declarar uma variável chamada "tempo" do tipo inteiro de 8 bits sem sinal:

```
unsigned int tempo;
```

ou:

```
int tempo;
```

Outro aspecto importante da declaração de variáveis é o local onde elas são declaradas.

A importância do local onde a variável foi declarada relaciona-se diretamente à acessibilidade ou não de outras partes do programa a essa variável. Isto se denomina regra de escopo da linguagem.

Basicamente, uma variável pode ser declarada em três pontos distintos do programa:

- No corpo principal do programa: as variáveis declaradas no corpo principal do programa (fora de qualquer função, inclusive da função **main()**) são chamadas de globais, porque podem ser acessadas de qualquer ponto do programa;
- Dentro de uma função: as variáveis declaradas dentro de uma função (incluindo a função **main()**) somente podem ser acessadas de dentro da função em que foram declaradas. Isto significa que uma variável local somente existe enquanto a função está sendo executada. No momento em que ocorre o retorno da função, as variáveis locais são descartadas;

- Como um parâmetro formal de uma função: as variáveis declaradas como parâmetros formais de uma função são um tipo especial de variáveis locais. Este tópico será melhor explicado no capítulo 8 quando estudarmos as funções da linguagem C.

Em seguida temos um pequeno programa que demonstra a diferença entre variáveis locais e variáveis globais:

Exemplo 5.1

```
#include <16f627.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200, xmit=PIN_B2,rcv=PIN_B1)
/*
   Teste de variaveis locais e globais
*/
int somatorio; // variável global

void soma(int valor)
{
   int conta;  //variável local da função soma
   somatorio = somatorio + valor;
   printf("A soma de 0");
   for (conta=1; (conta<(valor+1)); conta++) printf("+%u",conta);
   printf(" eh igual a %u\r\n",somatorio);
}
void main()
{
   int conta;     // variável local da função main
   somatorio = 0;
   for (conta = 1; conta<20; conta++)
   {
         soma(conta);
   }
}
```

A execução deste programa no hardware de laboratório proposto gera a seguinte saída na tela do emulador de terminal serial:

```
A soma de 0+1 eh igual a 1
A soma de 0+1+2 eh igual a 3
A soma de 0+1+2+3 eh igual a 6
A soma de 0+1+2+3+4 eh igual a 10
A soma de 0+1+2+3+4+5 eh igual a 15
A soma de 0+1+2+3+4+5+6 eh igual a 21
A soma de 0+1+2+3+4+5+6+7 eh igual a 28
A soma de 0+1+2+3+4+5+6+7+8 eh igual a 36
A soma de 0+1+2+3+4+5+6+7+8+9 eh igual a 45
A soma de 0+1+2+3+4+5+6+7+8+9+10 eh igual a 55
A soma de 0+1+2+3+4+5+6+7+8+9+10+11 eh igual a 66
A soma de 0+1+2+3+4+5+6+7+8+9+10+11+12 eh igual a 78
A soma de 0+1+2+3+4+5+6+7+8+9+10+11+12+13 eh igual a 91
A soma de 0+1+2+3+4+5+6+7+8+9+10+11+12+13+14 eh igual a 105
A soma de 0+1+2+3+4+5+6+7+8+9+10+11+12+13+14+15 eh igual a 120
```

```
A soma de 0+1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16 eh igual a 136
A soma de 0+1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17 eh igual a 153
A soma de 0+1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18 eh igual a 171
A soma de 0+1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19 eh igual a 190
```

Antes de analisarmos o funcionamento do programa, vamos explicar algumas das novas declarações encontradas nele:

No início do programa, encontramos duas declarações:

```
#use delay(clock=4000000)
#use rs232(baud=19200, xmit=PIN_B2,rcv=PIN_B3)
```

Estas declarações são diretivas do compilador e possuem a seguinte função:

A diretiva **delay**, conforme já dito no capítulo anterior, é utilizada para informar ao compilador a velocidade do clock do processador em que será executado o programa.

A diretiva **rs232** é utilizada para ordenar ao compilador que gere código para comunicação serial utilizando uma interface serial assíncrona padrão rs232.

Podemos observar que os parâmetros da diretiva são: velocidade de comunicação em bps (baud=19200), pino de saída dos dados transmitidos (xmit) e pino de entrada dos dados recebidos (rcv). Maiores detalhes sobre esta diretiva serão vistos mais adiante neste livro.

Podemos verificar que o programa é composto de duas funções:

A função **soma**, encarregada de realizar a soma do número atual ao valor acumulado até ali, e **main**, que é a função principal do programa, sendo responsável pela contagem de 1 a 20 e a chamada da função **soma**.

Podemos ainda constatar a existência dos três tipos básicos de variáveis no programa anterior:

A variável global "somatorio", encarregada de acumular o valor total da soma dos números. Esta variável, por estar declarada no corpo principal do programa, ou seja, fora de qualquer função, pode ser acessada de qualquer ponto do programa, incluindo as funções **soma** e **main**.

Dentro da função **soma**, encontramos a variável "conta" que, em virtude de sua declaração dentro da função **soma**, é considerada local, isto é, somente pode ser acessada dentro da função na qual foi declarada.

Também encontramos um outro tipo de variável na função **soma** e que muitas vezes passa despercebido. Observe a declaração da função:

```
void soma(int valor)
```

Podemos fazer duas constatações básicas sobre a declaração da função **soma**: a primeira é que esta função não retorna nemhum valor, já que foi declarada como sendo do tipo **void**. A segunda é que esta função possui um parâmetro formal declarado. A função soma recebe o seu parâmetro pela variável do tipo inteiro "valor". Veja que esta variável constitui um tipo especial, pois apesar de ser declarada na função soma, efetivamente comporta-se como uma variável global.

Finalmente, na função **main** encontramos também uma declaração de variável:

```
int conta;
```

Este comando declara, dentro da função **main**, uma variável inteira chamada conta. Observe que esta também é uma variável de escopo local, já que está declarada dentro do corpo da função principal **main**.

Uma observação interessante a ser feita é que ambas as funções, **soma** e **main**, possuem uma variável local chamada "conta". Note, no entanto, que o conteúdo destas variáveis é diferente. Na prática, o compilador aloca posições distintas de memória RAM do PIC para acomodar as duas variáveis, mas isto depende da hierarquia de chamadas de função e alguns fatores de otimização que serão discutidos oportunamente neste livro.

Outra observação oportuna a ser feita: a variável "somatorio", por ter sido declarada no corpo principal do programa (ou seja, fora de qualquer função) pode ser acessada de qualquer ponto do programa, como pode ser constatado pela análise da função **soma** e da função **main**.

5.3.1. Variáveis Locais

Neste momento, é importante observarmos mais detalhadamente o funcionamento das variáveis locais na linguagem C.

Há pouco, definimos as variáveis locais como sendo aquelas declaradas dentro do corpo de uma função. Bem, isso não é inteiramente verdade.

De fato, o corpo de uma função é apenas uma especificação de uma definição mais ampla: variáveis locais são aquelas declaradas dentro de um bloco de código.

Isto é importante, pois podemos ter variáveis locais mesmo dentro de blocos de código pertencentes a outros comandos. Veja o exemplo seguinte:

Exemplo 5.2

```
int x;     // declara a variável local x
x = 5;     // atribui o valor 5 à variável x
if (x>2) // se x maior que 2 então
{
   int y;  // declara a variável local y
   /*
   o código seguinte atribui o valor lido da porta A à variável
   y e em seguida compara esse valor com 3. Se forem iguais,
   os pinos da porta B são zerados (output_b(0)) se forem dife-
   rentes, é colocado na porta B o valor igual a x+1
   */
   if ((y = input_a())==3) output_b(0); else output_b(x+1);
}
```

Este programa demonstra alguns dos conceitos abordados neste tópico: observe que "y" é uma variável de escopo local declarada dentro do bloco da declaração **if**. Esta variável somente existirá durante a execução do bloco de comandos da declaração **if** (caso a condição seja avaliada como verdadeira).

Observe também que a segunda declaração **if** demonstra uma outra característica da linguagem C que é permitir a atribuição em uma expressão relacional. No caso, o valor retornado da função **input_a()** é primeiramente atribuído à variável "y", em seguida o valor é comparado a 3.

5.4. Constantes

Além das variáveis, também as constantes de dados, devido à sua aplicação, não poderiam deixar de ser incluídas na linguagem C.

Constantes são valores (numéricos ou não) fixos e que não podem ser alterados pelo programa durante a sua execução.

De maneira geral, qualquer tipo de dado pode ser utilizado para definir uma constante, cabendo apenas ao programador representar o valor dela adequadamente. Assim, qualquer constante pode ser declarada simplesmente especificando o nome do identificador e o seu valor. Veja os exemplos seguintes:

```
const val1 = 10
const val2 = -5
const val3 = 55.12
const val4 = 'a'
const val5 = "teste"
```

As constantes são também utilizadas para impedir que uma função altere um parâmetro passado a ela. Maiores detalhes desta aplicação das constantes podem ser vistos no capítulo 8.

5.4.1. Códigos de Barra Invertida

Alguns dos caracteres ASCII não podem ser inseridos diretamente em uma constante *string*. Um exemplo claro é o caractere de retorno de carro (código 13 decimal), o caractere de nova linha (código 10 decimal), o caractere de aspas ("), apóstrofo ('), entre outros.

O motivo pelo qual cada um não poder ser inserido varia, mas vamos exemplificar o de retorno de carro (que é aquele inserido pela tecla ENTER) e o caractere de aspas ("): o primeiro não pode ser inserido diretamente pelo teclado, pois o retorno de carro é inserido no código-fonte, provocando um efeito indesejado. O segundo, mesmo estando facilmente disponível no teclado, não pode ser inserido diretamente na *string*, porque o compilador irá interpretá-lo como um sinal do término da *string*.

Para inserir estes e outros caracteres em uma *string*, existem os chamados códigos de barra invertida. Na tabela seguinte encontramos os códigos mais comumente utilizados:

Código	Caractere	Código ASCII
\yyy	Constante Octal yyy	-
\xyyy	Constante hexadecimal yyy	-
\0	Nulo (null)	0
\a	Campainha (BEL)	07h
\b	Retrocesso (backspace)	08h
\t	Tabulação horizontal (TAB)	09h
\n	Linha nova (line feed)	0Ah
\v	Tabulação vertical (Vertical TAB)	0Bh
\f	Avanço de formulário (form feed)	0Ch
\r	Retorno de carro (Return)	0Dh
\"	Aspas	22h
\'	Apóstrofo	27h
\\	Barra invertida "\"	5Ch

Tabela 5.3

5.4.2. Constantes Binárias, Hexadecimais e Octais

As constantes de bases numéricas diversas da decimal são de suma importância na programação orientada a microcontroladores.

A linguagem C prevê a capacidade de manusear dados numéricos nas bases binária, octal, decimal e hexadecimal. Para utilizar uma constante de base numérica diferente da decimal, podemos utilizar um dos seguintes códigos:

Valor	Base numérica
99	Decimal
099	Octal
0x99	Hexadecimal
0b10011001	Binário

Tabela 5.4

Vejamos um exemplo de utilização de constantes numéricas de outras bases:

```
int valor1;
valor1 = 50;     // atribui o valor 50 decimal à variável valor1
valor1 = 0x50;   // atribui o valor 50 hexadecimal à variável valor1
valor1 = 050;    // atribui o valor 50 octal à variável valor1
valor1 = 0b01010000;  // atribui o valor binário 01010000 à variável
valor1
```

5.5. Operadores

Toda linguagem de programação necessita de operadores e nesta matéria, podemos dizer que a linguagem C é realmente muito rica.

Em C, contamos com operadores aritméticos (adição (+), subtração (-), multiplicação (*), divisão (/), resto de divisão (%), incremento (++) e decremento (--)), lógicos e relacionais (maior (>), menor (<), igual (==), diferente (!=), AND (&&), OR (||)) e muitos outros.

Maiores detalhes sobre o conjunto de operadores da linguagem e o seu funcionamento podem ser vistos no capítulo 6.

5.6. Expressões

Expressões são elementos formados pela união de variáveis, constantes e operadores entre si. De maneira geral as expressões seguem as mesmas regras gerais das expressões algébricas em matemática.

Vejamos alguns exemplos de expressões em C:

Exemplo 5.3

```
custo = valor;
a = b + 1;
teste * 3;
```

Em C, é também possível a utilização de expressões condicionais, conforme o exemplo a seguir:

Exemplo 5.4

```
int x,y;
x = 10;
y = (x > 5) * 10;
// Como a condição é avaliada verdadeira, y será igual a (1) * 10 = 10.
// Se a expressão fosse falsa: y = (0) * 10 = 0.
```

Isto é possível porque em C, as expressões são avaliadas e resultam sempre em 0 (falso) ou 1 (verdadeiro).

Outra característica importante da utilização de expressões em C é a ausência de verificação de tipos. Suponha a seguinte expressão:

Exemplo 5.5

```
unsigned int8 a;
unsigned int16 b;
unsigned int32 c;
b = 300;
c = 100000;
a = c - b;
```

Não é difícil perceber que a variável "a", definida como sendo de 8 bits, não dispõe de capacidade para armazenar "c - b" que resulta no valor 99.700, ou seja, um valor de 17 bits. Este tipo de procedimento geraria um erro em outras linguagens, mas não em C.

C é uma linguagem muito maleável quanto à construção de expressões constituídas de tipos diferentes de dados.

A ferramenta que torna esta operação possível é a chamada **conversão de tipos**.

Assim, no programa anterior, o compilador realiza uma conversão de tipos, convertendo o resultado de 17 bits (99.700 decimal), que é efetivamente armazenado sob a forma de 32 bits, em um valor de 8 bits (116 decimal), tal qual demonstrado em seguida:

```
99700      = 00000000 00000001 10000101 01110100
```

Convertido para 8 bits:

```
116        =                    01110100
```

No próximo tópico veremos as regras que regem a conversão de tipos nas expressões em C.

5.7. Conversão de Tipos

Sempre que tivermos uma expressão composta por dados de tipos diferentes entre si, será aplicada a chamada conversão de tipos. As regras de conversão de tipos em C são basicamente três:

1) Em uma atribuição, o tipo do dado resultante da expressão é convertido no tipo de dado da variável que recebe a atribuição;

2) Todos os tipos **short int** e **char** são convertidos no seu tipo base imediatamente superior, ou seja, **int**;

3) Cada par de operandos de tipos diferentes é convertido no tipo base do tipo superior, o que é também chamado de promoção de tipos.

Nas atribuições, a conversão depende basicamente de a variável-alvo da atribuição ser de maior ordem ou não que o tipo resultante da expressão.

A tabela seguinte representa a forma de conversão para cada tipo de dado envolvido em uma atribuição.

Tipo da expressão	Tipo da Variável Alvo	Resultado
short int	char, int, long int ou int32	Somente o bit 0 é utilizado; os outros bits são zerados
char, int, long int ou int32	short int	O bit 0 do resultado da expressão é armazenado na variável; os outros bits são descartados.
char ou int	long int ou int 32	O byte LSB da variável; assume o resultado da expressão; a parte MSB é zerada.
long int ou int32	char ou int	Os 8 bits menos significativos do resultado da expressão são armazenados na variável; os outros bits são descartados.

Tabela 5.5 (Continua)

Tipo da expressão	Tipo da Variável Alvo	Resultado
int32	long int	Os 16 bits menos significativos do resultado da expressão são armazenados na variável; os demais bits são descartados.
int32	float	O valor da expressão é truncado para o armazenamento na variável float; possível perda de dados.
float	short int	O bit menos significativo da parte inteira do resultado da expressão é armazenado na variável.
float	char ou int	Os 8 bits menos significativos da parte inteira do resultado da expressão são armazenados na variável.
float	long int	Os 16 bits menos significativos da parte inteira do resultado da expressão são armazenados na variável.
float	int32	A parte inteira do resultado da expressão é armazenada na variável.
signed	unsigned	Se o resultado da expressão for negativo, o resultado será maior que o módulo do resultado.
unsigned	signed	Se o resultado da expressão for maior que o extremo de faixa do tipo, o resultado será negativo.

Tabela 5.5

No caso de uma operação envolvendo dois tipos diferentes, o tipo de menor ordem será convertido no tipo de maior ordem, conforme esta seqüência:

Maior Ordem	Float
	int32
	long int ou int 16
	char, int ou int8
Menor Ordem	short int ou int1

Tabela 5.6

Vejamos alguns exemplos:

Exemplo 5.6

```
int x,y;
long z;
```

```
x = 5;
z = 1000;
y = x + z; // o resultado y será igual a 237 decimal
```

No exemplo anterior, a expressão "x + z" resulta em um tipo **long**, que é convertido no mesmo tipo de y (**int**), provocando a truncagem dos 8 bits mais significativos.

Exemplo 5.7

```
float a,b;
int c;
long d;
int32 e;
a = 5.5;
c = 10;
d = 330;
e = 100000;
b = ((e / d) + c) * a; // o resultado b será igual a 1721.5
```

Neste exemplo o compilador avalia as expressões pela ordem determinada pelos parênteses e traduzi primeiramente a expressão "e / d", convertendo a variável "d" no tipo **int32**. O resultado é somado à variável "c", sendo esta convertida no tipo **int32**, da mesma forma. Em seguida, o último resultado é multiplicado pela variável "a" do tipo **float**. Isto significa que o resultado anterior será primeiramente convertido no tipo **float** e em seguida multiplicado pela variável "a". O resultado final é atribuído diretamente à variável "b", pois ela é do mesmo tipo (**float**) que o resultado da expressão.

Além da conversão de tipos realizada pelo compilador, o programador pode forçar explicitamente uma expressão a ser de um determinado tipo. Esta operação chama-se modelagem (ou em inglês *CAST*) e segue esta forma:

```
(novo_tipo) expressão
```

Em que "novo_tipo", determina o tipo de dado (dentre os disponíveis em C) no qual a expressão seguinte será convertida.

De fato, a modelagem do dado constitui-se em uma operação, o que faz do *casting* um tipo de operador unário (já que envolve apenas 1 operando).

Vejamos um exemplo de utilização da modelagem de dados: suponha que se deseje dividir uma determinada variável inteira (por exemplo "x") por um outro valor inteiro (2, por exemplo), mas por algum motivo seja necessário que o resultado inclua também a parte fracionária. Se escrevermos simplesmente:

```
x / 2
```

Note, que pelas regras já estudadas, o resultado da expressão será também do tipo inteiro. Assim, o resultado será também sempre do tipo inteiro. Para podermos obter um resultado do tipo float, do modo desejado, poderíamos escrever:

```
(float) x / 2
```

Um erro muito comum em programação C é a escrita de uma expressão como esta:

Exemplo 5.8

```
long x;
int y;
y = 10;
x = y * 100;  // o resultado em x será 232 !!!!!
```

Este programa não produz o resultado esperado: a multiplicação de "y" (inteiro de 8 bits) por 100 (constante inteira de 8 bits) produz outro resultado de 8 bits, que será então convertido em um número de 16 bits e armazenado em "x".

Para efetuar a operação desejada, devemos primeiramente converter a variável "y" em 16 bits, em seguida multiplicá-la por 100. Veja como seria:

Exemplo 5.9

```
long x;
int y;
y = 10;
x = (long) y * 100;  // o resultado em x será 1000
```

Outro erro clássico em programação C é o do exemplo 5.10:

Exemplo 5.10

```
long x,y;
float z;
y = 11;
x = 2;
z = 10 + ( y / x );
// O resultado da expressão acima será igual a 15 e não 15,5 como se
// poderia esperar
```

Uma vez que as variáveis "y" e "x" são de tipos inteiros, a operação de divisão também será inteira, provocando o descarte da parte fracionária do resto da divisão.

Para alcançarmos o objetivo desejado no programa acima, temos de instruir primeiramente o compilador, de que a divisão (y / x) deve ser realizada no formato de ponto flutuante, veja o exemplo 5.11.

Exemplo 5.11

```
long x,y;
float z;
y = 11;
x = 2;
z = 10 + ((float) y / x );
// O resultado da expressão acima será igual a 15,5
```

5.8. Modificadores de Acesso

A linguagem C disponibiliza dois modificadores ou qualificadores de tipo usados para especificar a forma que o compilador utiliza para acessar o conteúdo das variáveis.

O primeiro modificador de tipo é **const**, que determina ao compilador que a variável seja tratada como uma constante, conforme já visto no item 5.4.

O segundo modificador é **volatile**, utilizado para determinar ao compilador que a variável por ele modificada pode ter seu conteúdo alterado a qualquer tempo, evitando assim que o compilador efetue otimizações de código que de outra forma poderiam provocar um comportamento errôneo do programa.

De maneira geral, o compilador CCS não necessita do uso do modificador **volatile**.

5.9. Modificadores de Armazenamento

Os modificadores de armazenamento são elementos especiais utilizados para controlar a forma como o compilador irá lidar com o armazenamento da variável.

Existem quatro modificadores definidos em C padrão: **auto**, **extern**, **static** e **register**.

O modificador **auto** é utilizado para definir o âmbito ou escopo da variável como local. Não é necessário utilizar esse modificador, porque, por definição, as variáveis em C possuem escopo local.

O modificador **extern** é utilizado para definir variáveis externas ao programa. Isto é muito utilizado na criação de programas grandes e complexos, utilizando diversos módulos separados e que são depois ligados por meio de um programa

especial chamado *linker*. O compilador CCS não permite variáveis do tipo **extern**, no entanto esta continua sendo uma palavra reservada da linguagem.

O modificador **register** é utilizado para instruir ao compilador que tente armazenar a variável diretamente em um registrador da CPU. Esse modificador não possui efeito no compilador CCS, já que todas as variáveis são armazenadas em registradores.

O último modificador, **static**, determina ao compilador que a variável ocupará uma posição permanente na memória.

As variáveis do tipo **static** funcionam como as variáveis globais no sentido de que não são destruídas ao término da execução da função na qual foram declaradas, e funcionam como variáveis locais no aspecto de que não são conhecidas fora da função em que foram originalmente declaradas. Além disso, as variáveis **static** são sempre inicializadas pelo compilador com o valor 0.

Variáveis **static** são utilizadas quando necessitamos manter o valor de uma variável local da função entre uma chamada e outra, o que facilita o trabalho de criação de bibliotecas de funções.

Um exemplo de utilização de variáveis **static** encontra-se em seguida:

Exemplo 5.12

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

int calcula(void);  // protótipo da função

 main()
 {
     int conta;
     for (conta=0; conta<5; conta++) printf("%d ", calcula());
 }

 int calcula(void)
 {
     static int valor;
     valor = valor + 10;
     return(valor);
 }
```

5.10. Alocação de Memória

Como já foi dito, cada vez que declaramos uma variável, o compilador irá reservar um ou mais endereços da memória RAM de forma a armazenar a variável.

Neste tópico iremos estudar a forma como são armazenadas as variáveis na memória do microcontrolador.

Como já foi dito, as variáveis do tipo inteiro são armazenadas sempre iniciando-se da parte menos significativa (LSB) para a parte mais significativa (MSB). Isto significa que se uma variável inteira de 32 bits contendo o valor 0x12345678 estiver armazenada no endereço 0x20 hexadecimal, teremos a seguinte distribuição na memória RAM:

Endereço	Conteúdo
0x20	0x78
0x21	0x56
0x22	0x34
0x23	0x12

Tabela 5.7

No caso das variáveis do tipo **float**, o formato utilizado é o ponto flutuante adotado pela Microchip.

Neste formato, são utilizados 4 bytes de memória, sendo divididos da seguinte forma:

Formato	eb	f0	f1	f2
IEEE754	sxxx xxxx	y.xxx xxxx	xxxx xxxx	xxxx xxxx
Microchip	xxxx xxxx	s.xxx xxxx	xxxx xxxx	xxxx xxxx

Tabela 5.8

Em que:

- eb é o expoente do número somado a 127, s é o bit de sinal e f0, f1 e f2 armazenam a parte fracionária da número. No caso do formato IEEE 754, y é o bit LSB do expoente eb.

Observe que o formato padronizado pelo IEEE difere do utilizado pela Microchip e adotado pelos compiladores CCS. Isto ocorre por uma questão de otimização, pois o o formato adotado pela Microchip permite uma execução mais veloz dos cálculos envolvendo dados do tipo ponto flutuante.

5.11. Redefinindo Tipos de Dados

A linguagem C provê ainda um comando que permite ao programador definir novos nomes para os tipos de dados existentes.

Este comando é o **typedef** e o seu formato geral é:

```
typedef tipo novo_nome;
```

Vejamos um exemplo de utilização do comando typedef:

Exemplo 5.13

```
typedef float fracao;

fracao variavel; // declara uma variável do tipo fracao (float)
```

5.12. Exercícios

1) Quais são os tipos básicos de dados na linguagem C padrão?

2) Qual é a diferença entre os tipos **short int** e **int1** no compilador CCS?

3) Qual é a diferença entre uma variável **signed** e outra **unsigned**?

4) Qual é a magnitude de armazenamento do tipo **long int**?

5) Suponha que o valor –1 é armazenado em uma variável do tipo **signed int** e em um determinado ponto do programa essa variável é atribuída a outra do tipo **unsigned int**. Qual é o valor efetivamente armazenado na última variável?

6) Qual é a diferença entre uma variável global e uma **static**?

7) No programa seguinte, qual o escopo (global/local) de cada uma das variáveis?

```
int teste;
int funcao1 (int a, int b)
{
   return a+b;
}
main()
{
   int valor;
   valor =5;
   teste = funcao1(valor,10);
}
```

8) Qual o resultado armazenado na variável "y" no programa seguinte?

```
int x,y;
x = 5;
y = (x==5) * 2;
```

9) Indique o resultado armazenado na variável "x" do programa seguinte:

```
int x,y;
long int z;
z = 0x1234;
y = z + 1;
x = y + 1;
```

10) No programa seguinte, qual o valor armazenado em "z"?

```
int x;
long y;
float z;
x = 110;
y = 300;
z = y / x;
```

11) Qual a utilidade da modelagem de dados ou *typecasting*?

Anotações

Capítulo 6

Operadores

Como já foi dito, a linguagem C possui uma gama de operadores, sendo possivelmente uma das linguagens com maior número de operadores disponível atualmente.

Esta característica é um dos pontos positivos da linguagem, já que C agrega aos operadores comumente encontrados nas linguagens de alto nível, os operadores encontrados freqüentemente em linguagens de baixo nível como o Assembly.

Podemos classificar os operadores da linguagem C em sete categorias principais: atribuição, aritméticos, relacionais, lógicos, lógicos bit a bit, de memória e outros.

6.1. Atribuição

A primeira categoria de operadores é também a mais utilizada. Em C, o operador de atribuição "=" é utilizado para atribuir um determinado valor a uma variável. Um exemplo de atribuição:

```
x = 10;
y = x;
```

Podemos verificar no programa anterior duas operações de atribuição: na primeira, foi atribuído o valor 10 à variável "x", na segunda, foi atribuído o valor de "x" (que é 10) à variável "y". Conclui-se então que ao final do programa, "y" será igual a 10.

Repare que a atribuição é sempre avaliada da direita para a esquerda e não é possível realizar uma atribuição no sentido inverso.

6.2. Aritméticos

São utilizados para determinar ao compilador que efetue determinada operação matemática em relação a um ou mais dados:

Operador	Ação
+	Adição
-	Subtração ou menos unário
*	Multiplicação
/	Divisão
%	Resto de divisão inteira
++	Incremento
--	Decremento

Tabela 6.1

Os operadores de adição, subtração, multiplicação e divisão dispensam comentários.

O operador % é utilizado para retornar o resto de uma operação de divisão inteira. Vejamos um exemplo:

5 / 2 = 2,5 em uma divisão real, ou 5 / 2 = 2, em uma divisão inteira, sendo o resto igual a 1.

Assim, o resultado de 5 / 2 é 2 e o resultado de 5 % 2 é igual a 1.

Os operadores de incremento e decremento são utilizados para somar 1 (incremento) ou subtrair 1 (decremento) de uma variável.

A forma geral para utilização destes dois últimos operadores é:

```
variável ++; ou variável --;
```

Ou ainda por meio de uma atribuição:

```
variavel_1 = variavel_2 ++; ou variavel_1 = variavel_2 --;
```

Observe que em ambos os casos, a atribuição ocorre da seguinte forma: o valor da variável "variavel_2" é armazenado em "variavel_1" e após isso, o conteúdo de "variavel_2" é incrementado ou decrementado.

No entanto, em C é também possível escrever:

```
variavel_1 = ++ variavel_2; ou variavel_1 = -- variavel_2;
```

Nestes casos, a operação de incremento/decremento é realizada antes da atribuição propriamente dita.

Vejamos um exemplo:

```
int x,y,z;
x = 0;
y = x ++;
z = ++ x;
```

Neste caso, após a execução dos três comandos, o valor da variável "x" será igual a 2, o valor da variável "y" será igual a 0 e o valor da variável "z" será igual a 2.

Observação importante: Não é possível utilizar os operadores de incremento ou decremento com variáveis ou tipos de dados complexos, tais como os tipos ponto flutuante.

Note que há uma diferença clara entre escrever "y = x + 1" e "y = ++x":

Ambas produzirão o mesmo resultado em "y", no entanto, no primeiro caso, somente a variável "y", alvo da atribuição, é alterada. Já no segundo caso, tanto "y", como "x" são alteradas !

6.3. Relacionais

São utilizados em testes condicionais para determinar a relação existente entre os dados:

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

Tabela 6.2

Não há muito o que falar sobre estes operadores, já que o seu funcionamento é idêntico ao que todos estudamos na disciplina de matemática e que utilizamos no nosso dia-a-dia.

Repare que os operadores relacionais podem ser utilizados para construir expressões condicionais como já visto no capítulo anterior.

6.4. Lógicos Booleanos

Os operadores lógicos ou booleanos são utilizados para realizar conjunções, disjunções ou negações entre elementos em um teste condicional. Os operadores lógicos somente podem resultar em um dos valores: verdadeiro ou falso.

Operador	Ação
&&	AND (E)
||	OR (OU)
!	NOT (NÃO)

Tabela 6.3

Os operadores relacionais são elementos de suma importância na construção de testes condicionais. Com esses operadores podemos relacionar diversas condições diferentes em um mesmo teste lógico.

Vejamos um exemplo:

```
int x,y;
x = 10;
if (x>5 && x<20) y = x;
```

Como podemos verificar, a variável "y" somente será igual ao valor da variável "x" se o valor de "x" for maior que 5 **e** "x" for menor que 20. O que nos leva a concluir que ao final da execução do programa "y" será igual a 10.

Observe que para determinar o resultado da conjunção no comando **if**, o compilador procede da seguinte forma: primeiramente avalia a expressão "x > 5", o que no presente caso resulta verdadeiro, já que "x", que vale 10, é maior que 5. Em seguida, procede à avaliação da expressão "x < 20", a qual também resulta verdadeira. Finalmente, o compilador verifica o resultado da conjunção (verdadeiro E verdadeiro) o qual é verdadeiro, executando assim a atribuição seguinte (y = x).

Um aspecto importante a ser observado é que em C, uma variável com valor igual a zero, será avaliada como falsa e se tiver um valor diferente de zero, será avaliada como verdadeira. Vejamos um outro exemplo:

```
int teste,teste_2;
teste = 0;
teste_2 = 0;
if (!teste) teste_2++;
```

Como podemos perceber, a avaliação de "!teste" será verdadeira, já que a variável possui valor zero e o teste verifica a negação da variável. Assim, no presente caso, a variável "teste_2" terminará com o valor 1.

Observação importante: Não é possível utilizar os operadores lógicos com variáveis ou tipos de dados complexos, tais como os tipos ponto flutuante.

6.5. Lógicos Bit a Bit

Os operadores lógicos bit a bit são utilizados para realizar operações lógicas entre elementos ou variáveis. No entanto, ao contrário dos operadores lógicos simples, os operadores lógicos bit a bit podem resultar em um valor da mesma magnitude dos elementos operados.

Operador	Ação
&	AND (E)
|	OR (OU)
^	XOR (OU exclusivo)
~	NOT (complemento de um)
>>	Deslocamento à direita
<<	Deslocamento à esquerda

Tabela 6.4

6.5.1. Operador & (And)

A operação lógica AND funciona da mesma forma que o operador booleano AND, mas com a diferença de que a operação é realizada separadamente para cada bit dos operandos. Vejamos um exemplo:

```
int v1,v2;
v1 = 100;
v2 = v1 & 15;
```

A operação AND representada ocorrerá da seguinte forma:

100 decimal = 0 1 1 0 0 1 0 0

AND (&)

015 decimal = 0 0 0 0 1 1 1 1

Resultado = 0 0 0 0 0 1 0 0

Isto significa que o valor armazenado em "v2" será igual a 4 decimal.

Podemos perceber que o operador lógico AND é uma excelente forma de desligar um ou mais bits de uma determinada variável, já que um bit 0 em um dos operandos fará com que o respectivo bit do resultado seja também igual a zero. Imagine que desejemos desligar os bits 5, 6 e 7 de uma variável chamada "portb". A forma de fazer isso poderia ser:

```
#byte portb = 6
portb = portb & 0b11100000;
```

6.5.2. Operador | (OR)

A operação OR, tal qual o operador AND, também trabalha de maneira similar ao seu equivalente booleano, com a diferença de que também aqui a operação é realizada para cada bit dos operandos. Vejamos um exemplo:

```
int v1,v2;
v1 = 0x20;
v2 = v1 | 0x04;
```

Esta operação OR será realizada da seguinte forma:

20 hexadecimal =	0 0 1 0 0 0 0 0
OR (|)	
04 hexadecimal =	0 0 0 0 0 1 0 0
Resultado =	0 0 1 0 0 1 0 0

Como podemos perceber, o resultado armazenado em "v2" será igual a 24 hexadecimal, e se analisarmos a operação, poderemos concluir que a operação OR constitui uma excelente forma de ligar bits de uma variável, já que os bits em um dos operandos permanecerão sempre ativados no resultado.

6.5.3. Operador ^ (XOR)

A XOR (exclusivamente OU) consiste em uma operação lógica entre dois, na qual o resultado somente será verdadeiro (nível lógico 1) se um e somente um deles for verdadeiro (nível 1). Ou seja, caso os operandos sejam iguais (0, 0 ou 1, 1), o resultado será falso (nível 0).

Operadores XOR são muito utilizados em funções de comparação de valores: se os bits dos operandos são iguais, o resultado é 0; se forem diferentes, o resultado é 1. Vejamos um exemplo do funcionamento do operador XOR em C:

```
int x,y;
x = 100;
y = x ^ 99;
```

Vejamos a execução da operação XOR:

```
100 decimal =  0 1 1 0 0 1 0 0
                 XOR (^)
 99 decimal =  0 1 1 0 0 0 1 1
 ------------------------------
  Resultado =  0 0 0 0 0 1 1 1
```

Podemos observar que apenas os bits diferentes entre os dois operandos resultaram em um valor 1. Os bits iguais resultaram em um valor 0. Se os operandos fossem iguais, o resultado seria igual a zero.

6.5.4. Operador ~ (NOT)

O NOT atua como operador de negação, ou em aritmética binária o complemento de um. Isto significa que o operador NOT inverte o estado de cada bit do operando especificado. Vejamos um exemplo:

```
int x,y;
long a,b;
x = 1;
a = 1;
y = ~ x;
b = ~ a;
```

Vejamos a operação do operador de complemento:

```
1 decimal =  0 0 0 0 0 0 0 1                    ( 8 bits, variável x)
------------------------------------------------------------------
Resultado =  1 1 1 1 1 1 1 0                    ( 8 bits, variável y)

1 decimal =  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1    ( 16 bits, variável a)
------------------------------------------------------------------
Resultado =  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0    ( 16 bits, variável b)
```

Concluímos então que a variável "x", de 8 bits, terminará o programa com o complemento do valor 1, ou seja, 254.

Já à variável "b" será atribuído o valor da negação de "a", de 16 bits, o que resulta no valor 65534.

6.5.5. Operadores de Deslocamento << E >>

Finalmente, temos ainda os operadores de deslocamento de bits à esquerda e à direita. O formato geral de uso destes operadores é:

- valor >> número de bits a deslocar à direita ou
- valor << número de bits a deslocar à esquerda

Vejamos um pequeno programa para demonstrar o funcionamento dos operadores de deslocamento:

```
int x, y, z;
x = 10;
y = x << 2;
z = x >> 1;
```

O funcionamento dos operadores de deslocamento é o seguinte:

Primeiramente é atribuído à variável "y" o deslocamento de dois bits à esquerda da variável "x". Vejamos o funcionamento desta operação:

10 decimal = 0 0 0 0 1 0 1 0

<<

0 0 0 1 0 1 0 0

<<

Resultado = 0 0 1 0 1 0 0 0

Observe que foram realizadas duas operações de deslocamento, sendo que o primeiro deslocamento resultou em 00010100 binário (20 decimal). Em seguida é realizado outro deslocamento, que resulta em 00101000 binário (40 decimal), sendo este valor atribuído à variável "y".

A próxima linha atribui à variável "z" o valor de "x" deslocado um bit à direita. Vejamos o seu funcionamento:

10 decimal = 0 0 0 0 1 0 1 0

>>

z = 0 0 0 0 0 1 0 1

Percebemos que o conteúdo da variável "x" (00001010 binário ou 10 decimal) é deslocado um bit à direita, resultando em 00000101 binário ou 5 decimal, sendo este valor atribuído à variável "z".

Com base nestas operações podemos verificar que no operador de deslocamento à esquerda, cada bit deslocado eqüivale a multiplicar o primeiro operando por 2. Já no operador de deslocamento à direita, eqüivale a dividir o operando por 2.

Observe que cada operação de rotação é traduzida pelo compilador em uma instrução assembly de rotação de bits (RLF ou RRF, conforme o caso).

Observação importante: Não é possível utilizar os operadores lógicos bit a bit com variáveis ou tipos de dados complexos, tais como os tipos ponto flutuante.

6.6. Memória

Os operadores de memória, também chamados de operadores de ponteiros, são elementos de grande importância na linguagem C. De fato, os ponteiros são considerados um dos pilares da linguagem C, pois permitem o acesso direto a qualquer endereço de memória do sistema. Existem dois operadores complementares para o acesso à memória:

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

Tabela 6.5

6.6.1. Operador &

O **&** é um operador unário utilizado para retornar o endereço de memória do seu operando. Isto significa que se escrevermos:

```
endereco_a = &a;
```

Teremos que a variável "endereco_a" conterá o endereço em que está armazenada a variável "a".

6.6.2. Operador *

Já o * é um operador unário utilizado para retornar o conteúdo da posição de memória endereçada pelo operando que o segue. Vejamos outro exemplo:

```
a = *endereco_a;
```

O que fará com que o valor armazenado no local apontado pela variável "endereco_a" seja atribuído à variável "a".

Veja em seguida um pequeno exemplo da utilização de ponteiros:

Exemplo 6.1

```
#include <16f628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)
int *endereco,x,y;
main()
{
   x = 5;
```

```
    endereco = &x;
    y = *endereco;
    printf("valor x= %d - endereco de x= %lx - valor y= %d",x,endereco,y);
}
```

Provavelmente a saída impressa deste programa será:

```
valor x= 5 - endereco de x= 26 - valor y= 5
```

O que significa que o valor de "&x" (endereço de "x") é igual a 0x26, ou seja, a variável "x" está localizada no endereço 0x26 da memória RAM do PIC.

Maiores detalhes sobre o funcionamento dos operadores de memória serão vistos quando estudarmos a operação dos ponteiros no capítulo 8.

6.7. Outros Operadores

Além dos operadores anteriormente citados, podemos encontrar ainda outros operadores não tão conhecidos em C:

Operador	Ação
?	Operador ternário condicional
,	Separador de expressões
.	Separador de estruturas
->	Ponteiro de elemento de estrutura
(tipo)	Operador de Modelagem de dado
sizeof	Retorna o tamanho da variável

Tabela 6.6

6.7.1. Operador?

O operador ternário "?" é utilizado para substituir uma expressão condicional baseada no comando IF e tem esse nome devido ao fato de ser composto sempre por três expressões. Sua forma geral é:

```
Variável = Expressão1? Expressão2: Expressão3
```

O que significa: avalie a expressão 1 e se ela for verdadeira, atribua à variável a expressão 2; caso contrário, atribua a expressão 3. Vejamos um exemplo:

```
int x,y;
x = 5;
y = x==7 ? 10: x+3;
```

O funcionamento deste programa é o seguinte: primeiramente é atribuído o valor 5 à variável "x". Em seguida, a expressão condicional é avaliada: se "x" for igual a 7, então "y" será igual a 10; caso contrário, "y" será igual a "x" mais 3. Como sabemos que "x" é igual a 5, teremos que "y" será igual a 8 (5+3).

Note que também é possível utilizar o operador "?" de uma forma pouco convencional, como neste outro exemplo:

Exemplo 6.2

```
#byte porta = 5
#byte portb = 6
int liga_led(void)
{
   porta = porta | 1; // ativa o pino 0 da porta A
}
int desliga_led(void)
{
   porta = porta & 254; // desliga o pino 0 da porta A
}
main()
{
        while (true)(portb & 1) ? liga_led() : desliga_led();
}
```

Neste pequeno programa, temos três funções: a principal **main()** e outras duas chamadas **liga_led** e **desliga_led**.

Na função principal, o programa permanecerá num *loop* infinito (**while (true)**) avaliando a expressão:

```
(portb & 1) ? liga_led() : desliga_led();
```

Se a avaliação de "portb & 1" resultar verdadeira, ou seja, diferente de zero, ou melhor dizendo: se o bit 0 da porta B estiver em nível lógico 1, será executada a primeira função **liga_led**; caso contrário, será executada a segunda função **desliga_led**.

6.7.2. Operador Vírgula

Outro operador pouco conhecido da linguagem C é o vírgula, utilizado para enfileirar duas ou mais expressões. A forma geral de utilização deste operador é:

```
variável = (expressão 1, expressão 2[,expressão x])
```

Note que as expressões são avaliadas da esquerda para a direita e a variável recebe o valor da última expressão avaliada.

Veja o exemplo seguinte:

```
y = (x=0, x+5);
```

Como podemos perceber, primeiramente a variável "x" assume o valor zero e em seguida, "y" irá assumir o valor de "x" mais 5, ou seja, ao final da avaliação da expressão, " y = 5 ".

Observe ainda que é possível colocar outras expressões não necessariamente relacionadas com a atribuição em si, como no exemplo seguinte:

```
z = (x = 0, y = 5, x - 2);
```

O compilador gera código para cada expressão individualmente, terminando por gerar o último código: "z = x – 2".

6.7.3. Operador Ponto

O operador ponto é utilizado em estruturas de dados como separador dos elementos e será estudado mais adiante, no capítulo 8.

6.7.4. Operador ->

Também chamado de operador seta, é utilizado para a função de ponteiro para uma estrutura de dados e será devidamente estudado no capítulo 8.

6.7.5. Operador de Modelagem de Tipo

Conforme já visto no capítulo anterior, a linguagem C dispõe de um operador unário destinado especificamente a forçar a conversão do operando especificado em um tipo de dado determinado. A isto, dá-se o nome de modelagem ou *typecasting*.

A forma geral do operador de modelagem é:

```
(tipo) variável
```

Onde "tipo" especifica o novo tipo de dado para o qual o conteúdo atual da variável especificada será convertido.

Observe que o conteúdo da variável não é alterado pelo operador, ao invés disso, aloca-se uma região de RAM para o armazenamento temporário do valor modificado.

6.7.6. Operador Sizeof

O operador **sizeof** é utilizado para retornar a quantidade de memória utilizada por uma determinada variável ou um determinado tipo de dado.

A principal aplicação deste tipo de operador é permitir ao programador controlar a ocupação de memória pelo programa, auxiliando assim na portabilidade do programa.

A forma geral do operador é:

```
sizeof variável ou
sizeof (tipo de dado)
```

6.8. Associação de Operadores

Para facilitar a vida do programador, a linguagem C inclui ainda uma outra característica que é a abreviação de operadores em atribuições e funciona da seguinte forma.

Normalmente, as operações de atribuição mais freqüentemente encontradas nos programas possuem a seguinte forma:

```
variável = variável (operando) valor {ou variável}
```

Nestes casos, a abreviação de operadores permite economizar a Segunda escrita da variável, reduzindo o comando à seguinte forma:

```
variável (operando)= valor {ou variável}
```

Na tabela seguinte, podemos encontrar os tipos de operadores abreviados admitidos em C.

Forma reduzida	Forma expandida
x += y	x = x + y
X -= y	x = x – y
x *= y	x = x * y

Tabela 6.7 (Continua)

Forma reduzida	Forma expandida
x /= y	x = x / y
x %= y	x = x % y
x &= y	x = x & y
x |= y	x = x | y
x ^= y	x = x ^ y
x <<= y	x = x << y
x >>= y	x = x >> y

Tabela 6.7

6.9. Precedência dos Operadores

Na tabela apresentada em seguida, podemos verificar a ordem de precedência de todos os operadores disponíveis na linguagem C. Por ordem de precedência entende-se a prioridade na avaliação de múltiplos operadores numa determinada expressão. Isto significa que operadores de maior ordem de precedência serão avaliados primeiramente em relação aos operadores de menor ordem.

É claro que o uso de parentesis em uma expressão, como podemos colher da tabela, permite aumentar a precedência de avaliação do conteúdo interno aos mesmos.

Ordem	Operador
Maior	( ) [ ] ->
	! ~ ++ -- . (tipo) * & sizeof
	* / %
	+ -
	<< >>
	<<= >>=
	== !=
	&

Tabela 6.8 (Continua)

Ordem	Operador
	^
	|
	&&
	||
	?
	= += -= *= /=
Menor	,

Tabela 6.8

No caso de operadores de mesma prioridade ou precedência, a ordem de avaliação é a natural, ou seja, da esquerda para a direita.

6.10. Exercícios

1) Qual é a diferença entre o operador = e o operador ==?

2) Qual é o resultado da expressão (10 && 5)?

3) Qual é o resultado da expressão (10 | 5)?

4) Como é avaliada a expressão (x = (10+5) < 20)?

5) Qual é o valor de "x" ao término do programa apresentado em seguida?

```
int x,y;
y = 256;
y += 1;
x = y >> 1;
```

6) Qual é o resultado da expressão (10 % 3 + 1)?

7) Qual é o valor de "y" ao término do programa seguinte?

```
long int x;
int y,z;
z = 100;
y = 10;
x = z * y + z && 1;
y = (int) x;
```

8) Qual é o valor armazenado nas variáveis "y" e "z" no programa seguinte?

```
long int x;
int y,z;
x = 0x1234;
y = x;
z = x>>8;
```

Capítulo 7

Declarações de Controle

As declarações ou comandos de controle são uma parte muito importante de uma linguagem de programação, e a linguagem C dispõe de um excelente conjunto de comandos.

Podemos classificar as declarações de controle em duas categorias básicas:

- Declarações de teste condicional: são utilizadas para testar determinadas condições / variáveis e executar um código para cada caso. A linguagem C dispõe de dois tipos de declarações condicionais: o comando **if** e o comando **switch**;
- Declarações de estrutura de repetição: são utilizadas para provocar a execução de um bloco de comandos enquanto uma determinada condição for verdadeira. Em C dispomos de três declarações de repetição: **for**, **while** e **do-while**.

Além destas declarações de controle, a linguagem C apresenta ainda a declaração **goto**, utilizada para provocar o desvio incondicional do programa.

7.1. Comando If

O comando **if** já foi visto anteriormente neste livro, portanto agora vamos analisar em maior profundidade o seu funcionamento.

De maneira geral, o comando **if** (ou "se" em português) é utilizado para executar um comando ou bloco de comandos no caso de uma determinada condição ser avaliada como verdadeira. Opcionalmente, é também possível executar outro comando ou bloco de comandos no caso da condição ser avaliada como falsa.

A forma geral do comando **if** é:

```
if  (condição) comandoA; { else comandoB;}
```

Nesta forma ele pode ser um comando ou declaração qualquer da linguagem C, ou um bloco de comandos na seguinte forma:

```
if (condição)
{
   comandoA1;      // bloco de comandos para a condição verdadeira
   comandoA2;
   ...
} else
{
   comandoB1;      // bloco de comandos para a condição falso
   comandoB2;
   ...
}
```

Observe que, em ambos os caso, a cláusula **else** é opcional, não sendo obrigatória para o funcionamento do comando.

O princípio de funcionamento do comando é muito simples: se a condição for verdadeira, será executado apenas o comandoA (ou o bloco de comandos A). Caso a condição seja avaliada como falsa, então será executado apenas o comandoB (ou o bloco de comandos B). Um detalhe muito importante a ser observado é que nunca, no mesmo teste, os dois comandos ou blocos de comando serão executados, apenas um ou outro.

Exemplo 7.1

```
char letra;
int x;
letra = 'A';
...
if (letra=='A') x=0; // caso a variável letra seja igual 'A', então x=0;
if (letra=='B') x=5; // caso a variável letra seja igual 'B', então x=5;
...
```

Um aspecto muito importante da linguagem C e que muitas vezes provoca confusões ao programador iniciante é que a condição a ser testada não precisa necessariamente envolver operadores relacionais ou lógicos. De fato, somente é necessário que a expressão possa ser avaliada como verdadeira ou falsa. Assim, é possível escrever o seguinte programa em C:

Exemplo 7.2

```
int x,y;
x = 5;
if (x) y = x;
```

Veja que a condição a ser testada é simplesmente o valor da variável "x". Se a condição for verdadeira, ou seja, se "x" for diferente de zero, então a atribuição "y = x" será executada. Caso "x" seja avaliada como falsa, ou seja, caso "x" seja igual a zero, então a atribuição não será executada.

É possível ainda encadear, ou aninhar, diversos comandos **if** um dentro do outro. Veja a forma geral em seguida:

```
if (condição1) comandoA;
   else if (condição2) comandoB;
           else if (condição3) comandoC;
                   ...
```

Neste caso, será primeiramente avaliada a expressão "condição1", se verdadeira, será executado o "comandoA", caso falsa, será avaliada a "condição2", caso seja verdadeira, será executado o "comandoB", caso seja falsa, será avaliada a "condição3" e se esta for verdadeira, será executado o "comandoC".

No exemplo seguinte, o programa faz uma leitura da porta A do microcontrolador e em seguida, utilizando uma estrutura de declaração **if/else,** testa sucessivamente o valor da variável "x" (lida da porta A), até encontrar um valor igual ao da variável, executando o comando **output_high** correspondente que fará setar um determinado pino da porta B do microcontrolador.

Exemplo 7.3

```
#include <p16f877.h>
#use delay(clock=4000000)
#fuses INTRC,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP

int x;

main()
{
   while (true)
   {
           x = input_a();
           output_port_b(0);
           if (!x) output_high(pin_b0); else
           if (x==1) output_high(pin_b1); else
           if (x==2) output_high(pin_b2); else
           if (x==3) output_high(pin_b3); else
                   ouput_high(pin_b4);
   }
}
```

Suponha que apenas o pino RA1 esteja em nível lógico 1 e os outros pinos da porta A estejam em nível lógico 0.

A execução deste programa fará com que seja atribuído o valor decimal 2 (lido da porta A) à variável "x". Em seguida, o primeiro comando **if** verifica se o valor da expressão (!x) é verdadeiro. Como "x" é igual a 2, então a expressão será avaliada como falsa e a próxima declaração **if** será avaliada. Novamente, o resultado da avaliação será falso, já que "x" é diferente de 1, fazendo com que a próxima declaração **if** seja analisada. Agora, como "x" é igual a 2, a avaliação da expressão será verdadeira e o comando **output_high**(pin_b2) será executado, fazendo com que o pino RB2 seja setado. Os outros pinos da porta B permanecerão em nível 0. Observe que o próximo **if** não é avaliado.

7.2. Comando Switch

Em alguns casos, como na comparação de uma determinada variável a diversos valores diferentes, o comando **if** pode tornar-se um pouco confuso ou pouco eficiente.

A declaração **switch** permite a realização de comparações sucessivas como a anterior, de uma forma muito mais elegante, clara e eficiente. Vejamos então o formato geral da declaração **switch**:

```
switch (variável)
{
   case constante1:
          comandoA;
          ...
          break;
   case constante2:
          comandoB;
          ...
          break;
   ...
   ...
   default:
          comandoZ;
          ...
}
```

Observe que o valor da variável é testado contra as constantes especificadas pela cláusula **case**. Caso a variável e a constante possuam o mesmo valor, então os comandos seguintes àquela cláusula **case** serão executados. Caso o valor da variável não encontre correspondentes nas constantes especificadas pelas **case**, então os comandos especificados pela cláusula **default** são executados.

Repare que cada seqüência de comandos da cláusula **case** é encerrada por uma cláusula **break**. Caso esta cláusula seja omitida, então todos os comandos subseqüentes ao **case** especificado serão executados, até que seja encontrada uma outra cláusula **break**, ou seja, atingido o final do bloco **switch**.

A explicação para este comportamento é simples: uma seqüência de comandos após uma cláusula **case** não é um bloco de comandos e sendo assim, deve haver uma forma de especificar o término dessa seqüência, o que é feito pela cláusula **break**.

Uma alternativa ao código anterior utilizando o comando **switch** pode ser:

Exemplo 7.4

```
// Este exemplo utiliza o circuito da figura 1.3
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP

main()
{
      setup_adc_ports (no_analogs);
      while(true)
      {
          output_port_b(0);
          switch (input_a())
          {
             0 : output_high(pin_b0);
                 break;
             1 : output_high(pin_b1);
                 break;
             2 : output_high(pin_b2);
                 break;
             3 : output_high(pin_b3);
                 break;
             default:
                 output_high(pin_b4);
          }
      }
}
```

No programa anterior, o valor lido da função **input_a()**, que efetua a leitura da porta A do microcontrolador, será comparado às constantes 0, 1, 2 e 3. Caso o valor corresponda a uma das constantes, a respectiva cláusula **case** será executada. Caso o valor lido não encontre correspondentes, será executada a cláusula **default**.

Algumas características importantes da declaração **switch**:

- A declaração **switch** somente pode testar igualdades. Não são admitidos outros operadores relacionais ou lógicos como em **if**;
- Somente dados ordinais podem ser utilizados como constantes, ou seja, números inteiros de 1, 8, 16 ou 32 bits, ou ainda constantes de caractere;
- Não é permitido que, dentro do mesmo **switch**, duas **cases** tenham constantes iguais;

- As cláusulas **break** e **default** são opcionais, havendo situações em que a não-utilização da cláusula **break** permite a construção de estruturas **switch** ainda mais eficientes, como veremos em seguida.

7.2.1. Cláusula Break

Como já dito anteriormente, a cláusula **break** possui a função de encerrar uma seqüência de comandos de uma cláusula **case**.

No entanto, podemos deliberadamente omitir a cláusula **break** para fazer com que uma mesma seqüência de comandos seja executada para várias **case**.

No fragmento de programa seguinte, caso a variável "x" possua um valor entre 0 e 3, será executado o primeiro bloco de comandos, caso possua o valor 4, será executado o segundo bloco de comandos e caso seja outro valor, será executado o último bloco de comandos:

Exemplo 7.5

```
switch (x)
{
    case 0:
    case 1:
    case 2:
    case 3:
       // caso o valor de x esteja entre 0 e 3, executa os comandos daqui
       // até o próximo break
       ...
       ...
       break; // encerra os comandos de todas as case acima
    case 4:
       // caso o valor seja igual a 4, executa os comandos daqui até o
       // próximo break
       ...
       break;
    default:
       // caso seja diferente de 0,1,2,3 ou 4, executa estes comandos
       ...
}
```

É aí que percebemos o motivo de não definir um bloco de comandos (na forma tradicional utilizando { e }) para cada cláusula **case**, pois se para cada case fosse definido um bloco de comandos, então não seria possível escrever a cláusula case 1 anterior, pois não poderíamos iniciar um bloco de código e não fechá-lo antes de outro bloco.

Repare que normalmente não se utiliza uma cláusula **break** após a cláusula **default**, já que normalmente **default** é a última cláusula de um comando **switch**.

7.3. Estruturas de Repetição

Como já foi dito no início deste capítulo, as estruturas de repetição, ou laços de repetição, como são também chamadas, são estruturas especiais de comandos, construídas de forma a executar repetidamente um determinado comando ou bloco de comandos. Na verdade, como veremos mais adiante, uma estrutura não precisa executar virtualmente nenhum comando.

Existem três estruturas de repetição na linguagem C: "Para", "Enquanto" e "Faça - enquanto".

A primeira estrutura é baseada no comando **for** ("Para") e é basicamente utilizada para laços finitos de contagem, normalmente utilizando uma variável de controle da contagem.

A estrutura baseada no comando **while** ("Enquanto") é utilizada para repetição de um determinado conjunto de instruções enquanto uma condição for verdadeira.

Finalmente, a estrutura baseada no comando **do-while** ("Faça - enquanto") é muito similar à estrutura **while**, com a diferença de que aqui a condição é analisada no final do bloco de comandos.

Vejamos então cada estrutura mais detalhadamente.

7.3.1. Laço For

O laço **for** é uma das mais comuns estruturas de repetição, sendo a versão C considerada uma das mais poderosas e flexíveis dentre todas as linguagens de programação.

O formato geral do laço **for** é:

```
for ( inicialização ; condição ; incremento ) comando;
```

ou,

```
for ( inicialização ; condição ; incremento )
{
   //bloco de comandos;
   comando1;
   comando2;
   ...
}
```

Cada uma das três seções do comando for possui uma função distinta, conforme em seguida:

- **Inicialização**: esta seção conterá uma expressão válida utilizada normalmente para inicialização da variável de controle do laço **for**.
- **Condição**: esta seção pode conter a condição a ser avaliada para decidir pela continuidade ou não do laço de repetição. Enquanto a condição for avaliada como verdadeira, o laço **for** permanecerá em execução.
- **Incremento**: esta seção pode conter uma ou mais declarações para incremento da variável de controle do laço.

O funcionamento básico do comando é o seguinte: primeiramente a seção de inicialização do comando é executada, em seguida, a condição de teste é avaliada e caso seja verdadeira, é executado o comando ou bloco de comandos, em seguida a seção de incremento é executada e o laço é repetido (voltando a avaliar a condição de teste).

O comando **for** pode ser utilizado para repetir um comando ou bloco de comandos, no entanto é também possível não executar nenhum comando e neste caso, o comando **for** possui apenas a função de retardar a execução do programa por um período de tempo.

Vejamos alguns exemplos de utilização do comando **for**:

O programa seguinte envia serialmente uma seqüência de números de 0 até 10:

Exemplo 7.6

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

main()
{
   int conta;
   for (conta=0; conta<=10; conta++) printf("%u\r\n", conta);
}
```

7.3.1.1. Outras Formas do Laço For

Como já dissemos no início do tópico anterior, o laço **for** possui uma grande flexibilidade na sua forma de construção e uso.

Esta flexibilidade é devido ao fato de que é permitida a utilização de qualquer expressão válida em C, mesmo não sendo relacionada com o laço de repetição em si, em quaisquer das seções do comando.

De fato, podemos criar laços muito interessantes e pouco usuais pela manipulação correta dos argumentos do comando **for**.

A primeira aplicação é na criação de laços de atraso (*delays*). Veja o seguinte fragmento de programa:

```
for (atraso=0; atraso<100; atraso++);
```

Este laço **for** não possui nenhum comando associado a ele, o que implica simplesmente que serão executadas 100 iterações do laço antes de o programa continuar a execução normalmente. Esta é uma forma muito eficiente de criar uma pequena rotina de atraso de tempo.

É possível também executar contagens regressivas com o comando for. O programa seguinte imprime na saída serial os números de 10 a 0 em ordem decrescente:

Exemplo 7.7

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

main()
{
   int conta;
   for (conta=10; conta>=0; conta--) printf("%u\r\n", conta);
}
```

Outra característica interessante do comando for é ser possível utilizar mais de uma variável para o controle do funcionamento do laço. No exemplo seguinte, demonstramos o uso de múltiplas variáveis de controle num mesmo laço:

Exemplo 7.8

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

main()
{
   int x,y;
   for (x=0, y=10 ; x<=10 ; x++, y--) printf("%u , %u\r\n",x,y);
}
```

O PIC programado com o código anterior imprime serialmente esta seqüência:

```
0 , 10
1 , 9
2 , 8
3 , 7
4 , 6
5 , 5
6 , 4
7 , 3
8 , 2
9 , 1
10 , 0
```

Observe que no exemplo anterior existem duas variáveis (apesar de somente uma ser testada na condição), sendo uma incrementada e outra decrementada a cada ciclo do laço.

Uma outra característica do comando for é ser possível utilizar uma condição de teste não relacionada à variável de controle do laço. No programa seguinte, construímos um laço de contagem que realiza uma contagem de 0 a 1000, mas caso o pino RB3 esteja em nível lógico 0, o laço terminará imediatamente e será impresso o valor final da contagem do laço:

Exemplo 7.9

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

main()
{
   long int x;
   for (x=0 ; (x<=1000) && input(pin_b3) ; x++);
   printf("%lu\r\n",x);
}
```

Este laço for não executa qualquer comando, mas permanece em contagem enquanto a variável "x" for menor ou igual a 1000 e o pino RB3 estiver em nível lógico 1.

É também possível criar um laço infinito, ou seja, um laço que será repetido indefinidamente. Veja o comando seguinte:

```
for (;;);
```

Alguns detalhes importantes sobre o comando for:

- Cada uma das seções de argumentos do comando for admite qualquer tipo de declaração válida da linguagem C;

- A seção de teste é sempre avaliada no início de cada ciclo do laço;
- Se a seção de teste for inicialmente falsa, nenhuma iteração do laço é executada;
- A seção de incremento é sempre executada ao final de cada ciclo do laço.

7.3.1.2. A Cláusula Break no Comando For

É possível também utilizar a cláusula **break** dentro de um laço for.

Nesse caso, **break** fará com que o laço seja prematuramente encerrado, independentemente de a condição de repetição do laço ser avaliada.

O mesmo programa do exemplo 7.9 pode ser escrito da seguinte forma utilizando a cláusula **break**:

Exemplo 7.10

```
#include <16F628.h>
#use delay(clock=12000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)


main()
{
   long int x;
   for (x=0 ; (x<=1000); x++) if (!input(pin_b3)) break;
   printf("%lu\r\n",x);
}
```

7.3.1.3. A Cláusula Continue no Comando For

Outra cláusula adicional que pode ser utilizada dentro de um laço **for** é a **continue**.

A sua finalidade é forçar o término do ciclo atual do laço, fazendo com que ocorra um novo ciclo.

O exemplo seguinte demonstra a utilização da cláusula **continue**. O código seguinte imprime pela saída serial os números entre 0 e 1000 que tenham o bit 4 igual a 1:

Exemplo 7.11

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
```

```
main()
{
   long int x;
   for (x=0 ; (x<=1000); x++)
        {
        if (!bit_test(x,4)) continue;
        printf("%lu\r\n",x);
   }
}
```

Observe que a cláusula **continue** não termina o laço como acontece com **break**. Ela apenas encerra o ciclo atual do laço.

7.3.2. Laço While

Outro tipo de laço disponível na linguagem é o comando **while**, já mencionado anteriormente neste livro e que possui a seguinte forma geral:

```
while (condição) comando;
```

ou ainda:

```
while (condição)
{
   comando1;
   comando2;
   ...
}
```

Como já foi visto, a filosofia de funcionamento do comando **while** é: primeiramente a condição é avaliada, caso seja verdadeira, então o comando ou o bloco de comandos associado é executado e a condição é novamente avaliada, reiniciando o laço. Caso o condição seja falsa, o comando ou bloco de comandos não é executado e o programa tem seqüência a partir da declaração seguinte ao bloco **while**.

Tal qual o comando for, a condição testada pelo **while** pode ser qualquer expressão da linguagem C que possa ser avaliada como verdadeira ou falsa.

Em seguida demonstramos como construir um laço de contagem simples utilizando o comando **while**:

```
int x;
while (x<=10) x++;
```

Assim como no comando **for**, não é preciso associar nenhum comando ao **while**. No exemplo seguinte, o programa permanece aguardando que o pino RB3 esteja em nível lógico 1, antes de prosseguir a execução:

Exemplo 7.12

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

main()
{
   while (!input(pin_b3));
   printf("O pino RB3 esta em nivel 1\r\n");
}
```

Algumas características do comando **while**:

- A condição de teste é avaliada sempre no início de cada iteração do laço;
- O comando ou bloco de comando, caso exista, será executado somente se a condição for verdadeira.

7.3.2.1. Break e Continue no Comando While

Também é possível utilizar as cláusulas **break** e continue com o comando **while**. Elas possuem a mesma função já explicada anteriormente.

Vejamos um exemplo do funcionamento de **break** e **continue** com o comando **while**:

Exemplo 7.13

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

main()
{
    port_b_pull-ups (true); // habilita pull-ups internos
    int x=0;                // declara a variável x como inteira de 8
                            // de 8 bits e a inicializa em 0
    while (x<30)            // enquanto x for menor que 30
    {
        // se o pino RB3 for igual a 0 sai do laço
        if (!input(pin_b3)) break;
        x++;                // incrementa x
        if (!(x%2)) continue;     // se o resto da divisão inteira de x por
                                  // 2 for 0 então termina o ciclo atual
        printf("%u\r\n" ,x);      // imprime o valor da variável x
    }
}
```

O exemplo 7.13 fará a contagem de 0 a 30 e imprimindo somente os valores de "x" que forem ímpares. Além disso, caso o nível lógico do pino RB3 seja igual a 0, o laço terminará prematuramente.

7.3.3. Laço Do-While

O último tipo de estrutura de repetição disponível na linguagem C é o comando **do** (em português "faça").

O comando **do** é utilizado juntamente com o comando **while** para criar uma estrutura de repetição com funcionamento ligeiramente diferente do **while** e **for** tradicionais.

De fato, a diferença entre a estrutura **while** tradicional e a estrutura **do-while** é que esta última realiza a avaliação da condição de teste no final de cada ciclo de iteração do laço de repetição, ao contrário do que já estudamos sobre o **while**, o qual realiza o teste no início de cada ciclo.

A forma geral da estrutura **do-while** é:

```
do comando while (condição);
```

ou,

```
do
{
   comandoA;
   comandoB;
   ...
} while (condição);
```

O funcionamento da estrutura **do-while** é o seguinte: o comando ou bloco de comandos é executado e então é avaliada a condição de teste, caso ela seja verdadeira, é iniciada uma nova iteração do ciclo; caso seja falsa, o laço é terminado.

Em seguida temos um exemplo de funcionamento do laço **do-while**:

Exemplo 7.14

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

main()
{
     port_b_pull-ups (true);      // habilita pull-ups internos
     int x=0;                     // declara e inicializa x
     do
```

```
    {
       x++;                        // incrementa x
       printf("%u\r\n",x);         // imprime o valor de x
    } while (input(pin_b3));       // enquanto o pino RB3 = 1
}
```

Este programa irá incrementar e imprimir o valor da variável "x" enquanto o estado lógico do pino RB3 for igual a 1.

Algumas características importantes do comando **do-while**:

- A condição de teste será avaliada sempre ao término de cada ciclo de repetição;
- Caso a condição seja inicialmente falsa, o comando, ou bloco de comandos, caso exista, será executado uma vez.

7.3.3.1. Break e Continue no Comando Do-While

Como os outros laços vistos neste capítulo, também o **do-while** aceita a utilização das cláusulas **break** e **continue**.

Veja uma modificação do exemplo 7.15 para encerrar prematuramente o laço no caso de atingido o valor 100:

Exemplo 7.15

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP

#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

main()
{
     int x=0;                      // declara e inicializa x
     do
     {
        x++;                       // incrementa x
        printf("%u\r\n",x);        // imprime o valor de x
        if (x>=100) break;         // se x maior ou igual a 100 sai do laço
     } while (input(pin_b3));      // enquanto o pino RB3 = 1
}
```

7.4. Comando Goto

A linguagem C dispõe ainda de outro comando de desvio incondicional do programa: **goto** (em português "vá para").

De fato, o desvio incondicional é um tanto incomum em uma linguagem de programação estruturada, mas existem algumas poucas situações em que o uso de goto pode acarretar um aumento na eficiência e velocidade do programa.

O funcionamento do comando **goto** é basicamente idêntico à instrução assembly *goto* disponível no conjunto de instruções dos PICs:

```
goto local;
```

Em que `local` é um rótulo para uma determinada posição no programa.

Com o uso do **goto**, é possível escrever estruturas de repetição ao estilo da linguagem BASIC e Assembly:

Exemplo 7.16

```
int conta = 0;                    // declara e inicializa a variável conta
loop:                             // define o rótulo loop
   conta++;                       // incrementa conta
   if (conta<=10) goto loop;      // se conta menor ou igual a 10 vai para
                                  // loop
```

Uma possível aplicação para o comando **goto** pode ser a saída imediata de uma estrutura profundamente aninhada.

7.5. Exercícios

1) No fragmento de programa seguinte, a condição analisada pelo comando if é verdadeira ou falsa?

```
int x = 0, y = 10;
...
if (x = 1) y = 0; else y = 5;
...
```

2) O programa seguinte aguarda o pressionamento de uma tecla e toma algumas decisões em relação ela. Qual a tecla em questão?

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)
main()
{
      char x;
      boolean y = 0;
      while (true)
      {
```

```
        x = getc(); // efetua a leitura de uma tecla
        if (x == 77)
        {
            y = !y;
            if (y) printf("Liga\r\n"); else printf ("Desliga\r\n");
        } else
        {
            y = 0;
            printf ("Desliga\r\n");
        }
    }
}
```

3) O programa seguinte permite visualizar alguns registradores internos do PIC por meio da linha serial. Qual(quais) tecla(s) devemos pressionar para visualizar o registrador TRISB?

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)
#byte fsr = 0x04
#byte indf = 0
main()
{
    int x=0;
    while (true)
    {
        switch (getc())
        {
            case 'p': switch (getc())
                      {
                          case 'a' :     x = 5;
                                         break;
                          case 'b' :     x = 6;
                      }
                      break;
            case 't': switch (getc())
                      {
                          case 'a' :     x = 0x85;
                                         break;
                          case 'b' :     x = 0x86;
                      }
                      break;
            default : x = 3;
        }
        if (x)
        {
            fsr = x;
            printf ("%x = %x\r\n",x,indf);
        }
    }
}
```

4) O programa seguinte imprime alguns valores na tela. Pressionando a tecla 5, quais são os valores impressos?

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)
main()
{
   int x,y;
   while ( (y = (getc()-48)) > 0)
           for (x = 0; x<=10; ++x) printf ("%u x %u = %u\r\n",y,x,x*y);
   printf ("Fim\r\n");
}
```

Capítulo 8

Tipos e Dados Avançados

Além dos tipos de dados já estudados, a linguagem C permite ainda outros tipos de dados de maior complexidade.

Neste capítulo vamos estudar outros tipos de dados mais complexos como os ponteiros, matrizes, strings de caracteres, estruturas, uniões e finalmente as enumerações, ou tipos enumerados.

8.1. Ponteiros

Os ponteiros são um tipo de dado e uma característica muito poderosa da linguagem C. O capítulo 3 já apresentou uma breve introdução aos operadores de ponteiros, porém sem apresentar quaisquer conceitos ou fundamentações para o seu uso, o que será feito neste tópico.

Podemos definir genericamente um ponteiro como sendo uma variável utilizada para guardar o endereço de outra variável, ou seja, um ponteiro é um apontador para outra variável.

A declaração seguinte pode ser utilizada para obter o endereço de uma variável inteira chamada "teste". Suponha que o conteúdo de "teste" seja igual a 255 decimal:

```
endereco_de_teste = &teste;
```

Desta forma, a variável "endereco_de_teste" passa então a ser um ponteiro para a variável "teste", armazenando o endereço físico da posição de memória ocupada por esta última. Repare o uso do operador "&".

Suponha que a tabela seguinte represente uma porção da memória RAM de um PIC no qual a variável "teste" ocupa a posição 0x20:

Endereço	Conteúdo
0x1F	0
0x20	255
0x21	30

Tabela 8.1

Podemos concluir que o valor da variável "endereco_de_teste" será igual a 0x20, que é o endereço de "teste".

Lembre-se de que o operador "&" vai retornar o endereço inicial de armazenamento da variável especificada. Isto é importante quando utilizamos variáveis de 16 bits ou mais, quando então o operador vai retornar o endereço de armazenamento do LSB da variável.

É possível ainda modificar o conteúdo da variável "teste" indiretamente, através do ponteiro recém-criado:

```
*endereco_de_teste = 10;
```

Esta declaração fará com que o valor 10 seja atribuído à posição de memória especificada por "endereco_de_teste". Observe a alteração na memória do microcontrolador:

Endereço	Conteúdo
0x1F	0
0x20	10
0x21	30

Tabela 8.2

Repare que utilizou-se agora o operador "*" para fazer referência ao conteúdo indicado pelo ponteiro.

Neste momento é importante observar que as variáveis do tipo ponteiro formam um novo tipo de dado e sendo assim, é necessário declará-las explicitamente.

A forma geral de declaração de uma variável ponteiro é:

```
tipo *nome_da_variável
```

Note que para declarar uma variável ponteiro, é necessário indicar o tipo de dado para o qual a variável irá apontar. Isto é necessário porque a linguagem C prevê toda uma gama de operações matemáticas envolvendo ponteiros

e essas operações são diretamente influenciadas pelo tamanho do dado referenciado pelo ponteiro.

Como exemplo de declaração de um ponteiro, vejamos um programa que engloba os exemplos anteriores:

```
int *endereco_de_teste;
int teste;
teste = 255;
endereco_de_teste = &teste;
*endereco_de_teste = 10;
```

Observe que este programa cria uma variável ponteiro inteira chamada "endereco_de_teste" e também uma variável inteira chamada "teste".

A seguir, é atribuído o valor 255 decimal à variável "teste" e em seguida o endereço desta variável é atribuído ao ponteiro "endereco_de_teste".

Dizemos que agora a variável "endereco_de_teste" aponta para a variável "teste".

Em seguida, atribui-se o valor 10 ao endereço apontado pela variável "endereco_de_teste". Resultado, a variável "teste" passa a armazenar o valor 10 decimal.

8.1.1. Operações com Ponteiros

É possível utilizar ponteiros em diversas operações da linguagem C como em expressões relacionais, atribuições, expressões aritméticas, etc. No entanto, em relação às operações aritméticas a linguagem C somente permite a realização de operações de adição e subtração e ainda assim, tais operações ocorrem de uma forma diferenciada em relação aos outros tipos de dados.

Devemos ter em mente que a principal finalidade de um ponteiro é apontar a posição na memória de um tipo de variável específico. Isto significa que a linguagem C deve utilizar mecanismos de forma a garantir o funcionamento do ponteiro, mesmo após uma operação envolvendo-o.

Tomemos por exemplo o programa seguinte:

Exemplo 8.1

```
#include <16f628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)


main()
{
```

```
    long int *endereco_de_teste, teste, outra;
    teste = 5;
    outra = 100;
    printf("Teste de ponteiros\r\n");

    endereco_de_teste = &teste;
    printf("Endereco do ponteiro: %Lx\r\n",endereco_de_teste);
    printf("Conteudo do endereco: %Lu\r\n",*endereco_de_teste);

    endereco_de_teste++;
    printf("Endereco do ponteiro: %Lx\r\n",endereco_de_teste);
    printf("Conteudo do endereco: %Lu\r\n",*endereco_de_teste);
}
```

Ao executarmos este programa, teremos os seguintes valores impressos na tela:

```
Teste de ponteiros
Endereço do ponteiro: 2d
Conteudo do endereco: 5
Endereço do ponteiro: 2f
Conteudo do endereco: 100
```

O valor 2d é o endereço hexadecimal da variável "endereco_de_teste" antes do incremento e o valor 2f é o endereço hexadecimal da mesma variável após o seu incremento.

Por que o valor aumentou em dois se foi realizado somente um incremento?

Muito simples: o tipo de dado base do ponteiro, naquele caso, é um inteiro longo, que ocupa por padrão 2 bytes de memória. Assim, ao executarmos um incremento do ponteiro, o compilador adiciona 2 ao valor do ponteiro, de forma a garantir que o novo endereço aponte para uma eventual nova variável inteira longa.

Observe que o conteúdo do primeiro endereço é o valor 5, que foi armazenado na variável "teste" e após o incremento do ponteiro, o conteúdo do novo endereço apontado é 100, que é o valor armazenado na variável "outra".

Isto significa que após o incremento do ponteiro, ele passou a apontar para uma variável diferente da qual foi atribuído inicialmente. Este tipo de operação pode ser muito útil na manipulação de dados, tabelas e matrizes, como veremos mais adiante.

De maneira geral, o incremento e o decremento de um ponteiro variam de acordo com o espaço em bytes ocupado pelo tipo base do ponteiro. Utilize a tabela 5.2 da página 60 para referência ao espaço ocupado por cada tipo de dado.

8.1.2. Tópicos Importantes Sobre Ponteiros

Antes de partir para o estudo de outros tipos de dados avançados da linguagem C, é importante observar alguns tópicos sobre o uso e funcionamento dos ponteiros na linguagem C:

- Nunca se esqueça de inicializar um ponteiro antes de utilizá-lo. Este é um erro comum, extremamente danoso ao programa e em alguns casos, de difícil depuração. Lembre-se que um ponteiro não inicializado pode apontar para qualquer região da memória (no caso dos PICs, somente memória RAM) e a utilização de um ponteiro nestas condições pode provocar alterações em outras variáveis ou registradores do microcontrolador;
- Observe com atenção redobrada as atribuições envolvendo ponteiros. Muitas vezes o programador desatento atribui o valor de uma variável (e não o seu endereço) a um ponteiro. Neste caso, teremos um ponteiro apontando para um endereço incorreto de memória, podendo provocar diversos efeitos e erros no programa;
- Os compiladores CCS C não suportam ponteiros para funções. Apesar de isto ser possível na linguagem C tradicional, por limitações da arquitetura do computador, não é possível implementar tais ponteiros nesses compiladores.

8.2. Matrizes de Dados

O primeiro dos tipos de dados complexos a ser estudado é a matriz de dados.

Uma matriz é um conjunto de dados homogêneo referenciado por um identificador único. Por dados homogêneos entendem-se variáveis ou dados do mesmo tipo. Assim, uma matriz será um conjunto de variáveis do mesmo tipo e referenciadas por um único identificador.

A linguagem C suporta a criação desde matrizes unidimensionais até matrizes multidimensionais.

No nosso cotidiano estamos acostumados a trabalhar com diversos tipos de matrizes: unidimensionais, como, por exemplo, listas (uma lista de nomes), bidimensionais (como um tabuleiro de xadrez, ou uma planilha de dados) e as multidimensionais com três ou mais dimensões.

Para declarar uma matriz na linguagem C, utilizamos os colchetes [ ]:

```
tipo de dado nome_da_matriz [ tamanho ]
```

`tipo de dado` é especificador do tipo de dado utilizado em cada um dos elementos da matriz. Podemos criar matrizes com qualquer dos tipos de dados disponíveis em C, incluindo as estruturas, que serão estudadas mais adiante neste capítulo. A única exceção, no caso do compilador CCS, é que não é possível criar matrizes de bit.

`nome_da_matriz` é o nome utilizado como identificador da matriz. Todas as referências à matriz serão realizadas pelo uso deste identificador.

`tamanho` é o número de elementos que pode ser armazenado na matriz.

Veja o exemplo em seguida:

```
int nota [40];  // define uma matriz de 40 números inteiros
char nome [20]; // define uma matriz de 20 caracteres
long outra [5]; // define um matriz de 5 números longos
```

Vejamos a representação gráfica da matriz "outra":

outra [0]
outra [1]
outra [2]
outra [3]
outra [4]

Tabela 8.3

O primeiro elemento é acessado pela declaração "outra [0]" e o último elemento, pela declaração "outra [4]".

Em C, o primeiro elemento de uma matriz é sempre o de índice zero.

O programa seguinte pode ser utilizado para inicializar os dados de uma matriz "nota":

Exemplo 8.2

```
int nota [40];
int indice;
for (indice = 0; indice < 40; indice++) nota[indice]=0;
```

A matriz "nota" criada anteriormente é um exemplo de uma matriz unidimensional ou lista, pois possui somente uma dimensão ou índice. Mais adiante veremos que é também possível criar matrizes multidimensionais em C.

Alguns detalhes importantes sobre matrizes em C:

- Uma matriz pode ocupar muita memória RAM. Podemos facilmente calcular a quantidade de memória RAM ocupada por uma matriz através da fórmula:

  Quantidade de memória = tamanho do tipo base * quantidade de elementos

  Isto significa que a matriz "nota" irá ocupar 1 byte * 40 = 40 bytes de RAM.

- Ausência de checagem de limites: devido às suas características de proximidade com a linguagem assembly, C não possui checagem de limites de matrizes.

  Isto significa que se escrevermos o seguinte código:

```
teste long int [10];
teste [10] = 300;  // este elemento está além do fim da matriz
```

  Em outras linguagens como BASIC e Pascal, esta operação geraria um erro de compilação, mas em C o programa será compilado normalmente.

  De fato, corremos um sério risco de modificar posições de memória utilizadas por outras variáveis, isso porque uma matriz de dez elementos terá como último elemento o de número 9 (lembre-se que o primeiro elemento é sempre 0). O que implica dizer que uma escrita no elemento de índice 10 é incorreta.

  Assim, o exemplo anterior iria modificar a memória RAM fora dos limites da matriz, com resultados imprevisíveis para o programa.

A linguagem C prevê ainda uma outra forma de inicialização de matrizes, chamada de inicialização explícita:

```
tipo de dado nome_da_matriz [ tamanho ] = { lista de valores };
```

Assim, a matriz nota utilizada nos exemplos anteriores poderia ser inicializada da seguinte forma:

```
int nota [5] = {0,0,0,0,0}; // inicializa todos os 5 elementos da matriz
int nota [5] = {4,9,7,3,6}; // inicializa a matriz com valores diferentes
```

Lembre-se que a seqüência de inicialização é a seguinte: o primeiro elemento da matriz (o de índice 0) é inicializado com o primeiro elemento da lista de valores (4), o segundo elemento da matriz é inicializado com o segundo elemento da lista de valores (9) e assim por diante até que o último elemento da matriz (no exemplo anterior o de índice 4) seja inicializado com o valor do último elemento da lista de valores (6).

É possível ainda inicializar uma matriz adimensional (sem tamanho definido). Veja o exemplo em seguida:

```
int teste [] = {6,7,8,9,4,5,6,7,8,9,2,3}
```

Este código cria uma matriz de 12 elementos chamada "teste" e inicializa-a com os valores especificados.

Lembre-se também que é possível inicializar os dados de uma matriz simplesmente utilizando o modificador **static**. Assim, em vez de escrever:

```
int nota [5] = {0,0,0,0,0};
```

poderíamos simplesmente escrever:

```
static int nota [5];
```

Com o mesmo efeito.

Lembre-se, no entanto, de que o uso do modificador **static** aloca permanentemente memória RAM para a matriz.

Atenção: A padronização ANSI da linguagem C prevê que as matrizes declaradas globalmente (fora de qualquer função ou bloco de comandos) sejam automaticamente inicializadas, caso o programador não o faça explicitamente, o que é chamado de inicialização por padrão. No entanto, o compilador CCS não implementa esta condição. Sendo assim, as matrizes globais devem ser inicializadas explicitamente, ou no código do programa.

Outra característica interessante da linguagem C é o fato de que a referência ao nome da matriz sem os colchetes para especificação do elemento é traduzida como um ponteiro para o primeiro elemento da matriz.

Isto significa que poderíamos nos referir ao **endereço** da matriz "nota" de duas formas diferentes:

```
&nota[0]
```

ou,

```
nota
```

Esta característica das matrizes é muito útil na passagem de matrizes como argumentos de funções, conforme veremos no próximo capítulo.

8.2.1. Matrizes Multidimensionais

Conforme já dissemos, é possível criar matrizes com mais do que uma dimensão.

As matrizes multidimensionais são estruturas formadas pela associação de matrizes unidimensionais. Todo o exposto até aqui para matrizes unidimensionais vale também para as matrizes multidimensionais.

No compilador CCS, podemos declarar matrizes multidimensionais de até cinco dimensões.

Para calcular o espaço ocupado em memória por uma matriz de múltiplas dimensões, basta multiplicar o número de elementos de cada dimensão entre si e o resultado multiplicar pelo tamanho em bytes do tipo base da matriz:

Quantidade de memória = tamanho do tipo base * (dimensão 1 * dimensão 2 * ...)

A forma mais simples de matriz multidimensional é a bidimensional, ou seja, formada por duas dimensões ou índices. Em C, declaramos uma matriz bidimensional da seguinte forma:

```
tipo de dado nome_da_matriz [ tamanho 1 ] [ tamanho 2 ]
```

ou,

```
tipo de dado nome_da_matriz [ linhas ] [ colunas ]
```

Vejamos um exemplo de declaração de uma matriz bidimensional:

```
int tabela [2] [5];
// declara uma matriz chamada tabela com 5 colunas e 2 linhas
```

A representação gráfica para esta matriz é:

tabela [0] [0]	tabela [0] [1]	tabela [0] [2]	tabela [0] [3]	tabela [0] [4]
tabela [1] [0]	tabela [1] [1]	tabela [1] [2]	tabela [1] [3]	tabela [1] [4]

Tabela 8.4

Para inicialização da matriz podemos utilizar:

Exemplo 8.3

```
int nota [2] [5];
int coluna,linha;
for (linha = 0; linha<2; linha++)
for (coluna = 0; coluna < 5; coluna++) nota[coluna] [linha]=0;
```

Uma outra forma alternativa para a inicialização da mesma matriz poderia ser:

```
int nota [2] [5];
int linha;
for (linha = 0; linha<10; linha++) *(nota+linha)=0;
```

Repare que neste exemplo utilizamos a característica de que o nome da variável, sem a especificação do elemento, aponta diretamente para o primeiro elemento da matriz. Além disso, como uma matriz aloca posições contíguas de memória, podemos nos referenciar ao primeiro elemento da matriz como sendo `*(nota)` e ao último, como sendo `*(nota+9)`.

Também é possível inicializar matrizes multidimensionais na própria declaração dela.

A matriz "nota" anterior, poderia ser inicializada da seguinte forma:

```
int nota [2] [5] = { 0,0,0,0,0,
                     0,0,0,0,0 };
```

ou ainda,

```
int nota [] [5] = { 0,0,0,0,0,
                    0,0,0,0,0 };
```

Note que a última forma definida anteriormente, apesar de ser aceita pelos compiladores ANSI, não é permitida pelos compiladores CCS e por isso não deve ser utilizada.

8.2.2. Strings de Caracteres

Quando construímos uma matriz unidimensional de caracteres, estamos na realidade criando um tipo de dado especial conhecido como string.

As strings não diferem muito dos outros tipos de matrizes encontradas na linguagem C, no entanto existem algumas características que precisam ser ressaltadas:

- As strings de caracteres são terminadas por um caractere nulo '\0';
- O tamanho total da matriz deve ser sempre 1 caractere maior que a quantidade de caracteres a ser armazenada na matriz. Isto significa que para uma string de 10 caracteres, a matriz deve ser declarada como tendo 11 caracteres (os 10 mais o caractere nulo ao final da string).

A declaração de uma string de caracteres pode ser feita da seguinte forma:

```
char teste [10];
char teste2 [13]={"Testando ..."};
char teste3 [] = {"Outro teste"};
char teste4 [4] [15] = {"Domingo","Segunda","Terca","Quarta"};
```

A primeira declaração é utilizada para criar uma string chamada "teste" com no máximo 9 caracteres válidos (o décimo deve ser o nulo).

Na segunda declaração, é criada uma string chamada "teste2" com capacidade de 13 caracteres, sendo inicializada com a constante string "Testando ...".

Repare que em C, uma constante string é definida entre aspas ("), já uma constante caractere é definida entre apóstrofos (').

Na terceira declaração, é criada uma string adimensional chamada "teste3", que é inicializada com a constante string "Outro teste". Observe que neste tipo de declaração, o compilador irá automaticamente dimensionar a matriz para o armazenamento dos caracteres da constante mais o terminador nulo, ou seja, a matriz "teste3" terá uma capacidade de armazenamento de 12 caracteres.

Na quarta e última declaração, demonstramos a criação de uma matriz de strings. No caso, será criada uma matriz bidimensional com quatro strings. A string de índice zero ([0]) é a "Domingo", a de índice [1] é a string "Segunda" e assim por diante.

8.2.3. Matrizes e Ponteiros

Como o leitor já deve ter observado, existe uma relação muito estreita entre matrizes e ponteiros na linguagem C.

De fato, já foi dito que a simples referência ao nome da matriz, sem qualquer especificação de índice, retorna o endereço (ou um ponteiro) para o primeiro elemento da matriz.

Mas este não é o único ponto comum entre matrizes e ponteiros. Outro aspecto interessante a ser observado é que em C, os ponteiros também podem ser indexados, tais qual uma matriz. Verifique o exemplo em seguida:

Exemplo 8.4

```
int *ponteiro, matriz [10];
// declara um ponteiro inteiro e uma matriz inteira com dez elementos
ponteiro = matriz;
// o ponteiro irá apontar para o primeiro elemento da matriz
matriz [0] = 10;    //o primeiro elemento da matriz é igual a 10
ponteiro[1] = 20;   //o segundo elemento da matriz é igual 20
```

8.3. Estruturas de Dados

Uma estrutura é um agrupamento de variáveis individuais, referenciadas por um nome comum.

Podemos imaginar uma estrutura de dados como um fichário, no qual o programador especifica os campos que farão parte das fichas. Desta forma, uma ficha irá agrupar diversas informações, normalmente relacionadas entre si, facilitando o acesso aos dados.

Em C, as estruturas de dados são criadas a partir dos outros tipos básicos de dados e até mesmo de outras estruturas.

A forma de criação de uma estrutura é dividida em duas etapas: primeiramente é definida a estrutura de dados à qual se atribui um identificador. Esse identificador passa, a partir daquele instante, a definir um novo tipo de dado na linguagem. Assim, o programador pode utilizar esse identificador para criar tantas variáveis de estrutura quantas quiser.

A forma geral de definição de uma estrutura em C é:

```
struct NOME
{
   tipo VARIÁVEL;
   tipo VARIÁVEL;
   ...
};
```

Uma vez definida a estrutura, o programador pode criar variáveis de estrutura utilizando a declaração:

```
struct NOME NOME_DA_VARIÁVEL;
```

Vejamos um exemplo de especificação e declaração de uma estrutura de dados para armazenar horas, minutos e segundos:

```
struct tempo
{
   int horas;
   int minutos;
   int segundos
};
struct tempo horario;
```

A seqüência apresentada criou uma estrutura de dados chamada "tempo" composta de três variáveis inteiras e em seguida criou uma variável de estrutura chamada "horario".

Repare que "tempo" não é uma variável e sim um especificador de tipo de dado.

Note que também é possível criar a estrutura e declarar variáveis no mesmo ato:

```
struct tempo
{
   int horas;
   int minutos;
   int segundos
} horario;
```

Neste programa, foi criada uma estrutura chamada "tempo" e após a sua definição, foi criada uma variável de estrutura chamada "horario".

Observe que a variável "horário" vai ocupar 3 bytes de memória RAM, já que ela se refere a uma estrutura composta de três elementos do tipo **int**.

Feito isso, o leitor pode se perguntar: mas como eu faço para acessar o conteúdo de uma variável dentro da estrutura?

A resposta é o operador ponto (.), que permite especificar o elemento da estrutura que se deseja acessar.

A forma geral para acesso ao elemento de uma estrutura é:

```
nome_da_variável_de_estrutura.nome_do_elemento
```

Assim, para inicializar com zero a variável "horas" dentro da estrutura definida anteriormente, podemos escrever:

```
horario.horas = 0;
```

8.3.1. Operações com Estruturas

Podemos realizar com uma variável estrutura, qualquer das operações já vistas para os outros tipos de dados da linguagem C.

Isto significa que é possível atribuir uma variável de estrutura a outra, conforme podemos observar no exemplo seguinte:

Exemplo 8.5

```
struct tempo
{
   int horas;
   int minutos;
   int segundos
} horario, alarme;

void main(void)
{
   horario.horas=0;        // zera as horas
```

```
   horario.minutos=0;      // zera os minutos
   horario.segundos=0;     // zera os segundos
   alarme = horario;       // os elementos da variável horário são atribuídos
                           // aos respectivos elementos da variável alarme
}
```

Neste momento, devemos observar um detalhe importante: o leitor pode imaginar se não seria possível inicializar todos os elementos de uma estrutura pela simples atribuição à variável de estrutura. Ou seja, para inicializar todos os elementos da variável "alarme" em 1, escreveríamos:

```
alarme = 1;
```

No entanto, isto não procede. A declaração anterior **não** irá atribuir o valor 1 a todos os elementos da estrutura representada pela variável "alarme".

Na realidade, uma atribuição conforme a anterior, inicializa apenas o primeiro elemento da estrutura (o elemento "horas") em 1. Os demais serão automaticamente zerados pelo compilador.

Por outro lado, podemos concluir que a declaração:

```
alarme = 0;
```

Irá surtir o efeito desejado, já que o primeiro elemento da estrutura vai receber o valor 0 e os demais serão automaticamente zerados pelo compilador.

Outro aspecto interessante das estruturas é o fato de que podemos utilizá-las como variáveis legítimas na construção de outros tipos de dados complexos.

Assim, podemos construir estruturas que têm como elementos outras estruturas, ou ainda, estruturas que têm como elementos matrizes, matrizes de estruturas, etc.

Observe que quando temos uma estrutura dentro de outra estrutura, dizemos que elas formam uma estrutura aninhada. A normatização ANSI especifica que esse aninhamento pode atingir no máximo 15 níveis, o que é algo extremamente grande, especialmente quando lidamos com algo tão restrito como um microcontrolador.

Em seguida temos um exemplo de utilização de estruturas em estruturas e também matrizes:

Exemplo 8.6

```
struct teste1
{
   int a;
```

```
   int b;
   long int c;
};
struct teste2
{
   int a2;
   float b2;
   int x [10];
   struct teste1 y;
} teste;
void main(void)
{
   teste.a2 = 5;
   teste.b2 = teste.a2 * 1.5;
   teste.x [1] = 50;
   teste.y.a = teste.a2;
   teste.y.c = 1000;
}
```

8.3.2. Ponteiros para Estruturas

A linguagem C permite também a criação de ponteiros para estruturas de dados.

Uma das principais aplicações deste tipo de ponteiro seria a passagem de estruturas por referência na chamada de funções.

A forma de declaração de um ponteiro para uma estrutura segue o mesmo princípio já visto até aqui para os outros tipos de dados. Assim, para declarar um ponteiro do mesmo tipo da estrutura principal do exemplo 8.6, podemos escrever:

```
struct  teste2  *ponteiro;
```

E então atribuir o endereço da variável de estrutura "teste" ao ponteiro com a declaração:

```
ponteiro = &teste;
```

Para acesso aos elementos da estrutura, através do ponteiro, podemos utilizar uma declaração como a seguinte:

```
(*ponteiro).a2 = 6;
```

No entanto, a linguagem C prevê um operador específico para o acesso a elementos de uma estrutura a partir de um ponteiro: o operador seta (->).

Assim, a mesma declaração anterior, poderia ser reescrita da seguinte forma:

```
ponteiro -> a2 = 6;
```

8.3.3. Campos de Bit

Os campos de bit são uma característica muito importante da linguagem C por permitirem o acesso direto a bits individuais em uma variável ou posição de memória do sistema.

Na verdade, podemos dizer que os campos de bit são uma extensão das estruturas de dados, uma vez que eles somente podem ser definidos dentro de uma estrutura de dados.

O formato geral de definição de um campo de bits é o seguinte:

```
struct nome_da_estrutura
{
   tipo nome1 : comprimento_em_bits;
   tipo nome2 : comprimento_em_bits;
   ...
} lista_de_variáveis;
```

Repare que a diferença entre um campo de bit e um campo normal de uma estrutura é que após o nome da variável (elemento), é especificado o seu comprimento em bits.

Nos compiladores CCS, o primeiro elemento da estrutura é armazenado no bit 0 (LSB) da posição de memória, o segundo elemento é armazenado no bit 1 e assim por diante.

Também é possível misturar campos de bit com variáveis normais em uma mesma estrutura:

```
struct nome_da_estrutura
{
   tipo nome1 : comprimento_em_bits;  // campo de bit
   tipo nome2 : comprimento_em_bits;  // campo de bit
   tipo nome3;  // um elemento normal (não bit)
   ...
} lista_de_variáveis;
```

A utilidade dos campos de bit fica evidente quando necessitamos acessar diretamente um elemento de hardware do dispositivo, algo bastante comum na programação de sistemas microcontrolados.

Vejamos um exemplo de uma estrutura para mapeamento do registrador INTCON , disponível em todos os PICs com capacidade de interrupção:

Exemplo 8.7

```
struct reg_intcon
{
   int rbif : 1;
   int intf : 1;
   int t0if : 1;
   int rbie : 1;
   int inte : 1;
```

```
   int t0ie : 1;
   int peie : 1;
   int gie  : 1;
} intcon;
#locate intcon=11
```

Repare que o tipo de todos os bits foi o mesmo (**int**), isto porque um campo de bit com comprimento de apenas um bit deve obrigatoriamente ser do tipo sem sinal (**unsigned**) e o compilador CCS (ao contrário do padrão ANSI) adota como padrão para o tipo **int** o modo sem sinal (**unsigned**).

Outro detalhe importante do exemplo 8.7 é a utilização da diretiva **#locate** para especificar o endereço de armazenamento da variável "intcon". Sem ela, a variável seria localizada na RAM (um registrador GPR qualquer), perdendo a sua função de acesso ao registrador físico do PIC.

Uma vez criada a estrutura e uma variável de estrutura, é muito simples operar um determinado bit do registrador:

```
intcon.gie = 1;  // habilita o controle global de interrupções
intcon.t0ie = 1; // habilita a interrupção do timer 0
```

Além das formas de declaração de estrutura com campos de bit vistas anteriormente, a norma ANSI permite também declarar campos de bit sem nome. Isto pode ser utilizado nos casos em que um ou mais bits não serão utilizados pelo programa, ou não estejam disponíveis no hardware. No entanto, o compilador CCS não permite a utilização de campos sem nome.

Um exemplo seria a declaração de uma estrutura para o registrador PCON, disponível na maioria dos novos PICs:

Exemplo 8.8

```
struct reg_pcon
{
   int por : 1;
   int bod : 1;
   int sem_uso: 6;  // 6 bits não utilizados
} pcon;
#locate pcon = 0x8E
```

Antes de encerrar o estudo dos campos de bit, é importante observar alguns detalhes sobre este tipo de dado:

- Não é possível obter o endereço de uma variável de campo de bit;
- Não é possível criar matrizes com variáveis de campo de bit;
- Não é possível ultrapassar os limites de um inteiro;
- Não é possível garantir que de um compilador para o outro a ordem de distribuição dos campos na memória seja idêntica, ou seja, se o primeiro campo será mapeado no bit MSB ou no bit LSB.

8.4. Uniões

As uniões são outro elemento importante da linguagem C e permitem que uma mesma posição de memória RAM seja compartilhada por diversos tipos de variáveis.

O formato geral da declaração de uma união é muito parecido com o das estruturas de dados já estudadas:

```
union nome_da_união
{
   tipo nome1;
   tipo nome2;
   tipo nome3;
   ...
} lista_de_variáveis;
```

Na declaração de uma união, o compilador cria uma variável grande o suficiente para armazenar o tipo de maior tamanho listado na união. Os demais tipos pertencentes serão armazenados sempre no mesmo endereço inicial da primeira variável da união.

Vejamos um exemplo de declaração de uma união:

Exemplo 8.9

```
union teste
{
   int32 variavel32;
   int16 variavel16;
   int   variavel8;
   int   vari8[4];
} vari_teste;
```

Na tabela seguinte podemos visualizar a alocação de memória para cada um dos elementos da união:

Elemento	Endereço			
	0	1	2	3
variavel32	LSB	...	...	MSB
variavel16	LSB	MSB		
variavel8				
vari8	vari8 [0]	vari8 [1]	vari8 [2]	vari8 [3]

Tabela 8.5

Pela tabela anterior, podemos perceber que todos os elementos da união são armazenados no mesmo endereço inicial.

A variável "variavel32" e a matriz "vari8" possuem cada uma um tamanho de 4 bytes, sendo portanto os maiores tipos da união.

Um valor armazenado no elemento "variavel32" ocupará todas as quatro posições de memória da união. Um valor armazenado no elemento "variavel16" ocupará somente os dois primeiros endereços da variável de união e assim por diante.

Isto significa que podemos acessar qualquer um dos bytes do elemento "variavel32", utilizando a matriz "vari8". Ou seja: suponha que armazenemos o valor 0x12345678 no elemento "variavel32":

```
vari_teste.variavel32 = 0x12345678;
```

Se verificarmos o valor armazenado, no elemento "vari8 [0]" encontraremos 0x12, no elemento "vari8 [1]" encontraremos 0x34, e assim por diante.

As uniões são uma forma muito elegante e eficiente de realizar conversões de tipo de variáveis de 32 ou 16 bits para variáveis de 8 bits ou vice-versa.

Repare ainda que é possível ter estruturas de dados como elementos de uma união. Assim, podemos construir as seguintes estruturas para realizar a conversão de 32 para 8 bits e vice-versa:

Exemplo 8.10

```
#include <16f628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,LVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)
struct parte
{
   int lsb;
   int lsb_alto;
   int msb_baixo;
   int msb;
};
union teste
{
   int32 variavel32;
   struct parte acesso;
} vari_teste;
void main( void )
{
   vari_teste.variavel = 0x12345678;
   printf ("Variavel 32 bits = %Lx\r\n",vari_teste.variavel32);
// imprime : Variavel 32 bits = 12345678
   printf ("LSB ... MSB = %x ", vari_teste.acesso.lsb);
   printf ("%x ", vari_teste.acesso.lsb_alto);
```

```
   printf ("%x ", vari_teste.acesso.msb_baixo);
   printf ("%x\r\n", vari_teste.acesso.msb);
// os comandos acima imprimem: LSB ... MSB = 78 56 34 12
}
```

8.5. Enumerações

Quando escrevemos um programa muito grande e complexo, é comum que o programador perca o controle sobre as variáveis e o significado dos seus estados.

Imagine um equipamento que utilize uma variável para determinar o modo de funcionamento: 0 - modo A, 1 - modo B, 2 - modo C, 3 - modo de espera e 4 - desligado.

À medida que o programa vai se tornando mais complexo e o tempo de escrita do programa aumenta, torna-se difícil distinguir os modos de funcionamento apenas pelos seus números: 0, 1, 2, 3 e 4. Torna-se necessário utilizar algum outro artifício para facilitar a visualização e utilização do programa.

A resposta da linguagem C a este problema são as enumerações: um conjunto de constantes inteiras utilizado para especificar os valores que uma variável pode assumir.

A sua forma geral é:

```
enum nome_da_enumeração
{
   lista_de_identificadores
} variável_enumerada
```

Assim, a enumeração para o problema de seleção de modos já descrito poderia ser:

```
enum enu_modos
{
   modo_a, modo_b, modo_c, modo_espera, modo_desligado
} modo;
```

Esta declaração cria uma lista de enumeração iniciando de zero (para o primeiro elemento, no caso "modo_a") até 4 para "modo_desligado".

Assim, podemos tranqüilamente utilizar os identificadores no lugar dos valores em qualquer expressão da linguagem envolvendo a variável enumerada. Assim, escrever:

```
modo = modo_b;
```

é o mesmo que escrever:

```
modo = 1;
```

É também possível alterar o valor de seqüência da enumeração simplesmente atribuindo um valor a um dos elementos da lista na declaração do tipo. Veja o exemplo:

```
enum enu_modos
{
   modo_a, modo_b, modo_c, modo_espera=5, modo_desligado
} modo;
```

Observe que agora a seqüência de valores da enumeração foi alterada: "modo_a" vale 0, "modo_b" vale 1, "modo_c" vale 2, "modo_espera" vale 5 e "modo_desligado" vale 6. Os elementos são enumerados com base no último valor de seqüência atribuído. Como o penúltimo elemento teve seu valor alterado, o último acompanhou a mudança.

8.6. Streams

O último tipo de dado avançado que vamos estudar é a *stream*, que em inglês significa uma corrente ou fluxo, e em termos computacionais trata-se de um fluxo ou canal de dados.

De fato, a stream é um tipo especial de dado utilizado para permitir o redirecionamento e manipulação de arquivos de dados.

No presente caso, as streams são utilizadas para permitir o redirecionamento da saída de algumas das funções de E/S serial para outros dispositivos.

O manual do compilador CCS traz um exemplo prático da utilização de streams para redirecionar o fluxo de dados de uma interface serial para outra. Veja um exemplo:

Exemplo 8.11

```
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT
#use rs232(baud=9600,xmit=pin_c6,rcv=pin_c7, stream = HOSTPC)
#use rs232(baud=4800,xmit=pin_b3, stream = saida)
main()
{
   while (true)
   {
        fputc ( fgetc(HOSTPC), saida);
   }
}
```

8.7. Exercícios

1) No fragmento de programa seguinte, qual é o valor armazenado na variável "teste"?

```
#byte var = 0x05
int *teste;
var = 10;
teste = &var;
```

2) No fragmento de programa seguinte, qual é o valor armazenado na variável "teste2"? Suponha que o endereço de va seja 0x20 e vb = 0x22.

```
long int va,vb,*teste1,*teste2;
va = 1000;
vb = 2000;
teste1 = &va;
teste2 = ++teste1;
```

3) No fragmento de programa seguinte, qual é o valor armazenado na variável "vy"?

```
long int vx;
int vy,*end;
vx = 0x1234;
vy = &vx + 1;
```

4) Quantos elementos possuem as matrizes seguintes e qual é o valor do último elemento?

a) `char palavra [] = { "Teste" };`

b) `char palavra [] = { 'T','e','s','t','e' };`

5) No fragmento de programa apresentado em seguida, qual é o valor armazenado na variável "vx"?

```
int nota [2] [5] = { 0,1,2,3,4,
                     5,6,7,8,9 };
int vx;
vx = nota [1] [4];
```

6) No fragmento de programa seguinte, qual é o valor armazenado na variável "vx"?

```
int nota [2] [5] = { 0,1,2,3,4,
                     5,6,7,8,9 };
int vx;
vx = * (nota + 4);
```

7) No programa seguinte, qual é a quantidade de memória total ocupada pela matriz "exemplo"? Qual é o valor impresso pela instrução **printf?**

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP,NOMCLR
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)
main()
{
   struct s_teste
   {
          int bit0 : 1;
          int bit1 : 1;
          int bit2 : 1;
          int bit3 : 1;
          int num;
          float x;
   };
   struct s_teste exemplo [10];
   int coluna,linha;
   for (linha = 0; linha<10; linha++)
   {
          exemplo[linha].bit0 = 0;
          exemplo[linha].bit1 = 1;
          exemplo[linha].bit2 = 1;
          exemplo[linha].bit3 = 0;
          exemplo[linha].num  = linha;
          exemplo[linha].x    = 0;
   }
   exemplo[5].x = 1 / (float) exemplo[9].num;
   printf ("%f\r\n",exemplo[5].x);
}
```

Anotações

Capítulo 9

Funções

Como já dissemos, a linguagem C permite modularizar uma seqüência de comandos de forma que possamos executá-la sempre que necessário, sem a necessidade de repeti-la fisicamente.

De fato, as funções C são muito parecidas com as sub-rotinas encontradas em outras linguagens como BASIC e Assembly.

Neste capítulo, vamos estudar em maiores detalhes o funcionamento, características e detalhes das funções na linguagem C.

9.1. Forma Geral

O formato geral (ANSI) de uma função é:

```
{tipo da função} nome_da_função ({parâmetros})
{
        //bloco de comandos
        comando1;
        comando2;
        ...
}
```

O **tipo da função** é usado para especificar o tipo de dado do resultado que a função vai devolver para o local onde foi chamada. A especificação do tipo de dado de retorno é opcional e caso seja omitida, o compilador assume que o dado retornado é do tipo inteiro.

O **nome_da_função** é um identificador utilizado para especificar o nome pelo qual será conhecida a função pelo resto do programa. Esse nome deve ser um identificador válido da linguagem C e não é possível utilizar um identificador já utilizado previamente.

Os **parâmetros** da função são utilizados para passagem de valores para que a função possa utilizá-los e efetuar os procedimentos para os quais foi

escrita. Eles consistem em uma lista de tipos e variáveis separados por vírgulas. A especificação de parâmetros é opcional, podendo o programador escrever funções que não utilizam qualquer parâmetro no seu funcionamento. No entanto, mesmo que não existam parâmetros, os parênteses devem ainda assim ser utilizados.

Um exemplo de declaração de função:

```
int quadrado ( int a )
{
   return a*a;
}
```

Note que muitos compiladores (o CCS não é um deles) suportam a forma original adotada por Kernighan & Ritchie para a declaração de uma função que é:

```
{tipo da função} nome_da_função ({variáves})
tipos;
{
        //bloco de comandos
        comando1;
        comando2;
        ...
}
```

Nesta forma de declaração, somente os nomes das variáveis são colocados na lista de parâmetros da função. Os seus tipos são definidos logo abaixo, no corpo da função.

Veja um exemplo da mesma função "quadrado" anterior, declarada pela forma clássica da linguagem:

```
int quadrado ( a )
int a;
{
   return a*a;
}
```

Lembre-se que esta forma de declaração não é suportada pelo compilador CCS, mesmo porque ela já não é mais utilizada e o seu uso não é incentivado pela norma ANSI.

A seguir, veja um exemplo da utilização de uma função definida pelo programador:

Exemplo 9.1

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

int soma(int a, int b)
{
   return a+b;
}

main()
{
      printf("1 + 1 = %d", soma(1,1));
}
```

Neste programa, foi definida uma função chamada **soma**, que irá retornar o valor da soma dos seus parâmetros "a" e "b". Observe que a palavra **int** colocada à frente do nome da função especifica que ela vai retornar um valor do tipo inteiro de 8 bits.

No corpo da função encontramos apenas o código **return a+b**, que é utilizado para provocar o retorno da função com o valor especificado, o que no caso será a soma do valor dos parâmetros "a" e "b".

No corpo principal do programa (**main()**) encontramos a função **printf** que em um dos seus argumentos realiza uma chamada à função **soma** com os parâmetros iguais a 1.

Veja que a função soma recebe os valores 1 e 1 como argumentos e durante a execução da função teremos "a = 1" e "b = 1".

A execução do programa resulta na seguinte saída:

```
1 + 1 = 2
```

Nos próximos tópicos vamos estudar em maiores detalhes os diversos aspectos da estrutura de uma função.

9.2. Regras de Escopo

No tópico 5.3 vimos que as variáveis podem ser declaradas em diversos pontos do programa, mas quando são declaradas dentro de uma função, encontramos uma situação especial.

Existem duas possibilidades distintas de declaração de variáveis dentro de uma função: na sua lista de parâmetros e no corpo da função.

Como já vimos no tópico 5.3.1, uma variável declarada no corpo de uma função é chamada de variável local, pois possuirá apenas uma abrangência (ou escopo) local, dentro da própria função em que foi declarada.

As variáveis locais não podem ser acessadas de fora da função, sendo acessíveis apenas no interior da função. Além disso, as variáveis locais são normalmente destruídas após o retorno da função (a destruição ou não do conteúdo da variável depende da otimização e consumo de memória avaliados pelo compilador durante a compilação do programa).

Outra possibilidade de declaração de uma variável é na seção de parâmetros da função.

Como já vimos, uma função pode conter parâmetros de chamada, sendo definidos na declaração da função.

Os parâmetros da função possuem um comportamento diferenciado, uma vez que se comportam como variáveis locais em relação à visibilidade, e ao mesmo tempo comportam-se como variáveis globais em relação à sua acessibilidade.

Isso quer dizer que uma variável declarada como parâmetro formal de uma função terá escopo global no sentido de poder ser acessada de qualquer ponto do programa, para que possa ser utilizada para efetuar a passagem de parâmetros para a função.

9.3. Passagem de Parâmetros

Uma função pode receber argumentos para a sua execução de duas formas distintas:

- Por valor: copiando o valor de um argumento diretamente para a variável encarregada de receber o parâmetro formal da função. Este foi o método utilizado até agora nos nossos programas.
- Por referência: na passagem por referência, não é o valor do argumento que é copiado para o parâmetro formal da função e sim o seu endereço. Desta forma, a função, recebendo o endereço do argumento, pode utilizá-lo internamente e inclusive alterar o seu valor. Daí resulta a principal característica da passagem por referência: a função pode alterar o valor do argumento passado a ela.

Possivelmente esta exposição não tenha esclarecido muita coisa ao leitor. Por isso, vejamos o exemplo seguinte:

Exemplo 9.2

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

float divide(float a, float b)
// Divide o parâmetro a pelo parâmetro b
{
     if (!b) return 0;  // verifica se b é 0, se verdadeiro retorna 0
     a/=b;              // caso b>0 a = a / b
     return a;          // retorna o valor de a
}

main()
{
     float a,b,c;
     a = 7;
     b = 3;
     c = divide(a,b);
     printf("a= %f , b= %f , c= %f",a,b,c);
}
```

O resultado impresso pela função **printf** será:

```
a= 7.000000 , b= 3.000000 , c= 2.333333
```

O funcionamento do programa pode ser descrito sucintamente da seguinte forma: as variáveis "a" , "b" e "c" da função **main** são declaradas e é atribuído o valor 7 a "a" e 3 a "b". Em seguida, é chamada a função **divide**, sendo passados como argumentos os valores das variáveis "a" e "b" respectivamente.

Durante a execução da função divide, o valor recebido pelo parâmetro "a" é alterado ("a/=b") e a função retorna o valor do parâmetro "a" alterado.

Em seguida, o valor retornado pela função é atribuído à variável "c" e é chamada a função **printf** que imprime o resultado na saída padrão.

Esta é uma legítima descrição de uma passagem por valor, já que podemos constatar que o valor armazenado nas variáveis locais da função **main,** passados como argumentos, não foram alterados pela função **divide**.

Mas como podemos realizar uma chamada por referência?

Simples: pela utilização de ponteiros no lugar dos parâmetros da função. Isto permite que em vez do valor do argumento seja passado o endereço dele. De posse do endereço do argumento, a função pode então, não somente ler o valor armazenado nele, como também alterar esse valor.

Vejamos então a realização de uma chamada de função com passagem de valores por referência:

Exemplo 9.3

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

void divide(float *a, float *b)

// Divide o parâmetro a pelo parâmetro b
{
      if (*b) *a/=*b;  // caso b>0 a = a / b
}

main()
{
      float a,b;
      a = 7;
      b = 3;
      divide(&a,&b);
      printf("a= %f , b= %f",a,b);
}
```

Este programa imprime o seguinte resultado:

```
a= 2.333333 , b= 3.000000
```

Observe que agora o valor da variável "a", que é local à função **main**, foi alterado: o resultado da operação de divisão realizada na função **divide** foi armazenado diretamente na variável "a" citada.

Por que a variável "a" foi alterada?

Simples: o valor passado para a função divide não foi o valor de "a" e sim o endereço da variável local "a" da função **main**. Isto permite que a função **divide** altere o valor apontado pelo endereço localizado no seu parâmetro "a".

9.4. Matrizes com Argumentos de uma Função

Em C, o compilador normalmente realiza chamadas de função com parâmetro, utilizando o método de passagem por valor, no entanto, no caso das matrizes, isto não ocorre.

A utilização de matrizes como argumentos de uma função baseia-se no princípio de que em C, a referência do nome da matriz, sem a indicação do(s)

elemento(s) interno(s), traduz-se num ponteiro para o primeiro elemento da matriz.

Assim, quando utilizamos uma matriz como elemento da lista de parâmetros de uma função, o compilador irá efetivamente realizar uma passagem por referência, mesmo que não especificado pelo programador.

Isto significa que quando utilizamos uma matriz como argumento de uma função, apenas um ponteiro para ela é passado para a função e não a matriz inteira.

Existem três formas de declarar uma função para receber matrizes como argumento:

1. Declarando a matriz na lista de parâmetros da função.

Neste caso, o programador deve declarar normalmente a matriz e suas dimensões na lista de parâmetros da função. Veja o exemplo:

Exemplo 9.4

```
void soma ( int matriz[20] , int valor)
// soma o parametro valor a todos os elementos da matriz
{
   int conta;
   for (conta=0; conta<20; conta++) matriz[conta]+= valor;
}
main()
{
   int valores[20];
   soma ( valores , 1);
}
```

2. Declarando a matriz sem elementos na lista de parâmetros da função:

Neste caso, o programador deve declarar a matriz sem os seus elementos, ou seja, uma matriz adimensional. Veja o mesmo exemplo anterior reescrito:

Exemplo 9.5

```
void soma ( int matriz[] , int valor)
// soma o parametro valor a todos os elementos da matriz
{
   int conta;
   for (conta=0; conta<20; conta++) matriz[conta]+= valor;
}
main()
{
   int valores[20];
   soma ( valores , 1);
}
```

3. Declarando um ponteiro para o mesmo tipo da matriz na lista de parâmetros da função.

   Neste caso, em vez de utilizar a matriz como elemento da lista de parâmetros da função, o programador utiliza apenas um ponteiro do mesmo tipo de dado da matriz e que irá receber o endereço dela. Veja o mesmo exemplo anterior reescrito:

Exemplo 9.6

```
void soma ( int *matriz , int valor)
// soma o parametro valor a todos os elementos da matriz
{
   int conta;
   for (conta=0; conta<20; conta++) matriz[conta]+= valor;
}
main()
{
   int valores[20];
   soma ( valores , 1);
}
```

Um aspecto importante deste exemplo é a demonstração de que a linguagem C permite indexar ponteiros, como acontece com as matrizes, conforme já estudado no capítulo anterior.

9.5. Estruturas como Argumentos de uma Função

Também é possível a passagem de elementos de uma estrutura ou mesmo estruturas inteiras para uma função.

A passagem de elementos de uma estrutura é feita da mesma forma que a passagem de variáveis comuns.

No caso da passagem de estruturas inteiras, apesar de ser possível a passagem da estrutura por valor (caso em que a estrutura inteira será duplicada na memória, já que os PICs não podem utilizar a pilha para armazenar argumentos de funções), é preferível a passagem por referência, utilizando um ponteiro para a estrutura, conforme já discutido no capítulo anterior.

9.6. Funções com Número de Parâmetros Variável

O padrão ANSI C prevê ainda a possibilidade de criar funções com número indefinido de parâmetros.

A forma básica de declaração de uma função com múltiplos parâmetros é:

```
tipo_de_retorno Nome_da_função (tipo par1, ...)
{
   comandos;
}
```

Um exemplo típico de função com múltiplos parâmetros é a função **printf** da linguagem C.

Observe que os compiladores CCS não suportam a criação de funções com número variável de parâmetros.

9.7. Retorno de Valores

Existem duas formas básicas de uma função retornar a execução ao local de onde foi chamada:

- Pelo término da execução de todos os comandos da função;
- Pela execução da declaração **return**.

O retorno da função pelo término da execução é uma forma bastante evidente.

Sempre que a execução de uma função alcança o seu último comando, ocorre o retorno da função, ou seja, o endereço de retorno é retirado da pilha e é carregado no contador de programa do processador, desviando a execução para o ponto seguinte à chamada da função.

Abaixo temos um exemplo de retorno de uma função pelo término da execução dos seus comandos:

Exemplo 9.7

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP

void muda_pino(int pino)
// Inverte o estado lógico de um pino do microcontrolador
{
      ouput_bit(pino,!input(pino));
}

main()
{
      muda_pino(pin_b3);  // inverte o nível lógico do pino RB3
}
```

Foi definida a função **muda_pino**, que possui como parâmetro o inteiro **pino**, que será utilizado para especificar o pino a ser invertido pela função.

Observe que o tipo de dado retornado pela função é **void**, ou seja, nenhum valor é retornado pela função.

No corpo da função encontramos uma chamada a outra função interna do compilador: **output_bit**, que é utilizada para colocar um determinado pino do microcontrolador (o primeiro argumento) em um determinado nível lógico (o seu segundo argumento).

Ao término da execução da função **output_bit**, a função **muda_pino** efetua o retorno, já que não existe mais código a ser executado.

Um aspecto importante sobre o retorno pelo término da execução da função é que neste tipo de retorno de função, ela normalmente irá retornar um valor 0. Para que ela retorne outros valores, é necessário utilizar a declaração **return**.

Como já foi dito, por padrão, uma função C que não possua declaração de tipo é considerada como sendo do tipo inteira, o que significa que o valor a ser retornado (**return**) pela função será do tipo inteiro.

É possível retornar outros tipos de valores, bastando selecionar adequadamente o tipo de dado na especificação de tipo da função.

Em seguida temos um exemplo de um programa que utiliza múltiplas declarações **return** em uma função cujo tipo de dado de retorno é **float**:

Exemplo 9.8

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

float divide(float a, float b)
// Divide o parâmetro a pelo parâmetro b
{
      if (!b) return 0;  // verifica se b é 0, se verdadeiro retorna 0
      return a/b;        // caso b>0 retorna a/b
}

main()
{
      float a,b;
      a = 7;
      b = 3;
      a = divide(a,b);
      printf("a= %f , b= %f",a,b);
}
```

Além de demonstrar a utilização de uma função com tipo de retorno diverso do inteiro, o exemplo anterior demonstra também que:

- É possível utilizar o mesmo nome para variáveis locais (como "a" e "b" na função **main**) e para as variáveis utilizadas como parâmetros formais de outra função ("a" e "b" na função divide);
- É possível utilizar diversas declarações **return** numa mesma função;
- Caso não seja explicitamente declarado o valor de retorno (através do uso da declaração **return**, ou equivalente), o valor retornado pela função será indefinido.

Um outro aspecto importante a ser destacado é que, caso não seja desejado o retorno de valores da função, ela deve ser declarada preferencialmente como **void**, o que acarreta, na maioria das vezes, uma maior eficiência e redução do código gerado.

9.8. Retorno de Valores em Funções Assembly

Quando utilizamos linguagem Assembly dentro do bloco de código de uma função, pode ser necessário efetuar o retorno da função a partir do próprio código Assembly.

Nestes caso, devemos utilizar a variável interna _RETURN_, definida pelo compilador, para o armazenamento do valor de retorno da função.

Vejamos um exemplo de uma função que soma seus dois parâmetros e devolve o resultado:

Exemplo 9.9

```
int soma(int a, int b)
{
#asm
      movf    a,w
      addwf   b,w
      movwf   _RETURN_
#endasm
}

main()
{
      int a,b;
      a = 7;
      b = 3;
      a = soma(a,b);
}
```

9.9. Protótipos de Função

Muitas vezes, quando um programa fica muito complexo e são utilizadas muitas funções, pode acontecer de ocorrer uma chamada a uma função que somente é definida mais adiante no programa.

Neste caso, o compilador sinaliza um erro, informando ao programador que a função é desconhecida (no compilador CCS será exibida a mensagem: *Undefined Identifier*).

Para solucionar estes problemas, existe em ANSI C (assim como em Pascal), a chamada prototipagem de função.

Protótipo de uma função é uma declaração prévia com o intuito de informar previamente ao compilador (antes da declaração real da função) o formato de declaração da função, incluindo o tipo de dado de retorno e os tipos e quantidade de parâmetros.

Um protótipo de função deve sempre ser declarado obedecendo aos mesmos tipos e parâmetros da declaração da função.

Normalmente, os protótipos de funções são declarados logo no início do programa e *sempre* antes de fazer referência às funções prototipadas.

Vejamos então um exemplo de prototipagem de função no compilador CCS:

Exemplo 9.10

```
#include <16F628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
#use rs232(baud=19200,parity=N,xmit=PIN_B2,rcv=PIN_B1)

void divide(float *a, float *b);  // protótipo de função divide
main()
{
      float a,b;
      a = 7;
      b = 3;
      divide(&a,&b);
      printf("a= %f , b= %f",a,b);
}

// declaração da função

void divide(float *a, float *b)

// Divide o parâmetro a pelo parâmetro b
{
      if (*b) *a/=*b;  // caso b>0 a = a / b
}
```

9.10. Recursividade

Uma função recursiva é aquela que possui dentro do seu código uma chamada para ela mesma.

Funções recursivas são muito úteis em diversos algoritmos de organização e pesquisa de dados, no entanto, devido ao pequeno tamanho da pilha de memória dos PICs, à incapacidade deles de armazenar dados na pilha e também ao uso intensivo da pilha nestes tipos de função, o compilador CCS não permite chamadas recursivas a funções.

9.11. Exercícios

1) É possível ter duas variáveis com o mesmo nome, estando uma situada em uma função e outra fora de todas as funções?

2) Como podemos declarar uma função que não retorne valores?

3) Qual a diferença entre passagem de parâmetros por valor e por referência?

4) Na função seguinte, quantos e quais erros podem ser observados?

```
void teste (int a, b);
{
   if (a) return (a); else return (a+b);
}
```

5) No fragmento de programa apresentado em seguida, qual é o valor armazenado na variável "v1"?

```
void func(int *a, int *b)
{
      int x;
      x = (*a + *b) / 2;
      *a = x;
}
main()
{
      int v1,v2;
      v1 = 7;
      v2 = 3;
      func (&v1, &v2);
      printf ("%u\r\n",v1);
}
```

6) Assinale Verdadeiro ou Falso:

( ) Em C é possível termos uma quantidade variável de parâmetros de função.

( ) As funções recursivas são admitidas em C, mas não são permitidas nos compiladores CCS.

( ) Por definição, as funções C retornam tipos char.

( ) Os parâmetros formais de uma função são, para todos os efeitos, variáveis locais.

( ) Na passagem de parâmetros por referência, o valor da variável é passado para a função.

( ) Por padrão, todos os tipos de dados, incluindo-se matrizes e estruturas, são passados por valor.

( ) Uma função do tipo VOID não retorna qualquer valor.

Capítulo 10

Diretivas do Compilador

Todo compilador, seja ele C, Pascal, BASIC, etc., possui uma lista de comandos internos que não são diretamente traduzidos em código. Esses comandos são, na realidade, utilizados para especificar determinados parâmetros internos utilizados pelo compilador no momento de compilar o código-fonte e são chamados de diretivas do compilador ou diretivas do pré-processador.

O compilador CCS C, como não poderia deixar de ser, possui uma grande quantidade de diretivas, que podem ser utilizadas para controlar diversos parâmetros, além de outros usos, conforme veremos em seguida.

#ASM {ASIS}
#ENDASM

Utilizada para inserir código assembly diretamente no programa em C. Os mnemônicos são inseridos entre a diretiva #asm e a diretiva #endasm.

Sintaxe:

```
#asm OU #asm ASIS
   código
#endasm
```

Existe uma variável interna do compilador, chamada _RETURN_, utilizada para especificar um valor de retorno para uma função escrita em assembly.

Normalmente, o compilador inserirá automaticamente o código necessário para efetuar a seleção do banco de memória correto para acesso aos registradores do dispositivo. No entanto, caso seja utilizada a forma #asm ASIS, o compilador não fará qualquer tipo de intervenção, ficando a cargo do programador efetuar essas seleções de banco.

Exemplo:

```
int teste_asm (int a, int b)
// soma em assembly os valores de a e b
{
   #asm
      movf    a,w ; copia o parâmetro a para w
      addwf   b,w  ; soma w com b e coloca em w
      movwf   _return_ ;armazena o resultado na variável de retorno
   #endasm
}
```

#BIT

A diretiva #BIT é utilizada para definir uma abreviação para um bit de uma constante ou variável já definida anteriormente.

Sintaxe:

```
#bit identificador = x.y
```

O identificador especificado é utilizado como abreviação para o bit "y" da variável ou constante "x".

O valor de "y" deve estar compreendido entre 0 e 7.

Esta diretiva cria essencialmente uma variável booleana mapeada diretamente no local especificado, o que é útil para efetuar o acesso direto a um bit de um registrador SFR ou GPR do PIC.

Exemplo:

```
#BIT T0IF = 0x0B.2  // define o identificador T0IF como sendo o bit 2 do
endereço 0x0B (INTCON)

T0IF = 0;  // desliga o bit 2 do endereço 0x0B
```

#BYTE

Utilizada para definir um identificador para uma variável ou constante.

Sintaxe:

```
#byte identificador = x
```

A diretiva #BYTE pode atuar de diversas formas diferentes:

- Caso o *identificador* seja uma variável já definida no programa, o compilador irá posicioná-la no endereço especificado por "x". O tipo da variável não é alterado;

- Caso o *identificador* não tenha sido definido anteriormente, o compilador irá criar uma nova variável inteira (8 bits) no endereço especificado por "x";
- Caso o argumento "x" seja uma variável, o compilador irá criar uma outra variável do mesmo tipo e identificada pelo *identificador*, localizada no mesmo endereço de "x".

O endereço atribuído pela diretiva #BYTE não é exclusivo e pode ser designado pelo compilador para o armazenamento de outras variáveis. Por isso, esta diretiva é basicamente utilizada para acesso aos registradores SFR dos PICs.

Exemplo:

```
#byte tmr0 = 0x01
#byte status = 0x03
struct
{
    boolean r_w;
    boolean c_d;
    int vago : 2;
    int dados : 4;
} porta_a;
#byte porta_a = 0x05;

porta_a.r_w = 0;
porta_a.dados = 10;
```

#CASE

Ativa a distinção entre caracteres maiúsculos/minúsculos.

Sintaxe:

```
#case
```

Note que o compilador, por padrão, não distingue caracteres maiúsculos de minúsculos, porém a norma ANSI determina que a linguagem C efetue a distinção entre caracteres maiúsculos e minúsculos.

__DATE__

Variável interna do compilador utilizada para fornecer a data de compilação do programa.

Sintaxe:

```
__DATE__
```

O formato da data é: DIA-MES-ANO. Exemplo: 01-DEZ-02

#DEFINE

Utilizada para substituir o identificador pelo texto especificado imediatamente depois dele.

Sintaxe:

```
#define identificador  texto
```

ou

```
#define identificador(x,y...) texto
```

A diretiva **#define** pode ser utilizada para criar substituições simples de texto, ou ainda para a criação de macrocomandos em linguagem C.

Se usada apenas para substituição de texto (primeira forma da sintaxe), a diretiva irá substituir todas as aparições do *identificador* pelo *texto* especificado.

Exemplo:

```
#define total 5
int matriz[total]; // cria uma matriz inteira de 5 elementos
```

Na segunda forma de utilização (segunda forma da sintaxe), esta diretiva irá permitir a criação de macrocomandos, ou seja, uma seqüência de comandos identificada pelo *identificador*. A cada aparição do *identificador* no programa, o compilador irá substituir o identificador pelo código da macro (*texto*).

A diferença entre esta forma e a anterior é que nesta, a diretiva permite a utilização de parâmetros junto ao *identificador*. Esses parâmetros são também substituídos durante a compilação do programa.

Exemplo:

```
#define hi(x)   (x<<4)   // cria a macro hi que lê somente o nibble
                         // superior do byte
...
int teste;          // declara a variável teste
teste = 0x90;
teste = hi(teste);        // teste será igual a 0x09
// é a mesma coisa que escrever teste = teste << 4;
```

#DEVICE

Define o nome do processador utilizado.

Sintaxe:

```
#device chip opções
```

Esta diretiva é utilizada para especificar ao compilador o dispositivo para o qual será compilado o programa.

O parâmetro *chip* especifica o nome do processador utilizado.

O parâmetro *opções* especifica uma das seguintes opções:

- *=5: utiliza ponteiros de 5 bits (disponível para todos os dispositivos);
- *=8: utiliza ponteiros de 8 bits (para dispositivos de 14 e 16 bits);
- *=16: utiliza ponteiros de 16 bits (somente dispositivos de 14 bits);
- ADC = x: Especifica o número (x) de bits a ser retornado pela função read_adc(), caso o dispositivo possua conversor A/D interno;
- ICD = TRUE: informa ao compilador para gerar código compatível com o ICD da microchip.

Observações:

1) É necessário sempre definir o tipo de processador utilizado. Essas definições encontram-se normalmente nos arquivos de cabeçalho de cada dispositivo, bastando ao programador incluir esses arquivos no seu programa.

2) Quando um tipo de dispositivo é especificado por esta diretiva, toda a configuração prévia de outros **#device** e **#fuse** é apagada.

3) Somente é possível definir um processador em cada programa.

Exemplos:

```
#device PIC16F628
#device PIC16F877 *=8
#device ICD = TRUE
#device PIC16F876 ADC=8
```

__DEVICE__

Variável interna do compilador que é definida pela diretiva **#device** anterior. O valor da variável **__device__** é igual aos últimos dígitos do nome do dispositivo.

Sintaxe:

```
__device__
```

Exemplo:

```
#if __device__ == 876
  setup_adc(all_digital)
#endif
```

#ERROR

Força o compilador a gerar uma condição de erro de compilação.

Sintaxe:

```
#error texto
```

Quando a diretiva **#error** é encontrada, o compilador aborta a compilação e apresenta uma mensagem com o *texto* indicado pelo programador.

É possível incluir uma macro no *texto* da mensagem, situação essa em que o compilador irá expandir o código da macro.

Exemplo:

```
#if __device__ == 675
#error Este programa não funciona no 12F675 !
#endif
```

__FILE__

Insere o nome do arquivo compilado no local desta diretiva no instante em que o programa é compilado.

Sintaxe:

```
__file__
```

#FUSES

Esta diretiva é utilizada para programar as opções da palavra de configuração (configuration word) dos PICs.

Sintaxe:

```
#fuses opções
```

Nesta tabela temos algumas das *opções* disponíveis geralmente na configuração dos PICs:

Opção	Descrição
LP	Oscilador LP
RC	Oscilador RC
XT	Oscilador XT
HS	Oscilador HS
INTRC	Oscilador interno *
INTRC_OSCOUT	Oscilador interno com saída de clock *
INTRC_IO	Oscilador interno, OSC1 e OSC2 como E/S *
WDT	Watchdog habilitado
NOWDT	Watchdog desabilitado
PROTECT	Proteção de código habilitada
PROTECT_75%	Proteção ligada para 75% da memória *
PROTECT_50%	Proteção ligada para 50% da memória *
NOPROTECT	Proteção de código desabilitada
PUT	Temporizador de Power-up ligado
NOPUT	Temporizador de Power-up desligado
BROWNOUT	Reset por queda de tensão habilitado *
NOBROWNOUT	Reset por queda de tensão desabilitado *
MCLR	Pino MCLR utilizado para RESET *
NOMCLR	Pino MCLR utilizado como entrada *
LVP	Programação em baixa tensão habilitada *
NOLVP	Programação em baixa tensão desabilitada (RB4 = E/S) *

* Esta opção não está disponível em todos os dispositivos.

Tabela 10.1

Lembre-se que as opções não especificadas são deixadas no padrão do dispositivo.

Para maiores detalhes, consulte o datasheet de cada dispositivo e também a opção VIEW > Valid Fuses no menu principal do compilador PCW.

Exemplos:

```
#fuses HS, NOWDT, PUT, NOBROWNOUT, NOMCLR, NOLVP
#fuses INTRC_IO, PUT
```

#ID

Permite programar os valores de identificação do chip (Ids).

Sintaxe:

```
#id número_de_16_bits
```

ou

```
#id número_4bits, número_4bits, número_4bits, número_4bits
```

ou

```
#id "nome do arquivo"
```

ou

```
#id CHECKSUM
```

Esta diretiva insere no arquivo hexadecimal gerado pelo compilador, os comandos necessários para programar os endereços de identificação do chip.

Exemplos:

```
#id 0x3657
#id 3,6,5,7
#id CHECKSUM
```

#IF
#ELSE
#ELIF
#ENDIF

Teste para compilação condicional.

Sintaxe:

```
#if expressão
   código
{#elif expressão
   código
#else
   código}
#endif
```

A diretiva **#if** é utilizada para inserir blocos condicionais de código, isto é, blocos que podem ou não ser compilados, de acordo com uma ou mais condições.

O funcionamento da diretiva é idêntico ao da declaração **if** já estudada, com a diferença de que a expressão especificada é avaliada em tempo de compilação e não em tempo de execução do programa. Isto significa que não podem ser utilizadas variáveis C na expressão, apenas identificadores internos criados pelo compilador ou pelo programador por meio da diretiva **#define**.

As diretivas **#elif** e **#else** são opcionais, no entanto toda diretiva **#if** deve ser terminada por um **#endif**.

Exemplos:

```
#if  __device__ == 675
#error Este programa não funciona no 12F675 !
#endif
```

#IFDEF
#IFNDEF
#ELSE
#ELIF
#ENDIF

Teste para compilação condicional.

Sintaxe:

```
#ifdef identificador
   código
{#elif
   código
#else
   código}
#endif

#ifndef identificador
   código
{#elif
   código
#else
   código}
#endif
```

As diretivas **#ifdef** e **#idndef** são utilizadas para inserir blocos condicionais de código, isto é, blocos que podem ou não ser compilados. No caso do **#ifdef**, o compilador irá verificar se o identificador está definido. No caso do **#ifndef**, o compilador irá verificar se o identificador não está definido.

As diretivas **#elif** e **#else** são opcionais, no entanto toda diretiva **#if** deve ser terminada por um **#endif**.

Exemplos:

```
#define teste
...
#ifdef  teste
...  // código a ser inserido para auxiliar na depuração do programa
#endif
```

#INCLUDE

Insere um arquivo texto externo a partir da posição atual.

Sintaxe:

```
#include <nome_do_arquivo>
```

ou

```
#include "nome_do_arquivo"
```

Esta diretiva é utilizada para inserir um arquivo texto externo especificado por nome_do_arquivo a partir da posição atual do arquivo.

Normalmente, utiliza-se esta diretiva para inserir arquivos de biblioteca e funções no código do programa atual.

Observe que se não for especificado o caminho completo do arquivo (diretório e subdiretórios), o compilador irá pesquisar na lista de diretórios do projeto para encontrar o arquivo.

No caso da primeira forma de sintaxe, o diretório em que está gravado o arquivo-fonte do programa será o último a ser pesquisado.

Caso se utilize a segunda forma de sintaxe, o diretório em que está gravado o arquivo-fonte do programa será o primeiro a ser pesquisado.

Exemplos:

```
#include <16f628.h>
#include "d:\projetos\biblio\eeprom.c"
```

#INLINE

Informa ao compilador que a função seguinte deve ser implementada na forma INLINE, ou seja, o código completo da função será inserido no local de cada chamada feita à função.

O uso desta diretiva é indicado nos casos em que é necessário minimizar a quantidade de acessos à pilha, ou quando o tempo gasto com desvios e retornos possa influenciar na performance do programa.

Note também que o uso desta diretiva fará com que aumente o consumo de memória, já que cada chamada de função irá inserir uma nova cópia do seu código no programa.

Exemplo:

```
#inline
swapbyte (int &a, int &b)
{
int temp;
temp = a;
a = b;
b = temp;
}
```

#INT_xxx

Informa ao programa que a função seguinte será utilizada para tratamento de interrupção.

Sintaxe:

#INT_AD	ADIF - Conversão A/D completa
#INT_ADOF	ADOF - Overflow da conversão A/D
#INT_BUSCOL	BCLIF - Colisão de barramento (I2C)
#INT_BUTTON	Pushbutton
#INT_CCP1	CCP1IF - Módulo CCP1
#INT_CCP2	CCP2IF - Módulo CCP2
#INT_COMP	CMIF - Comparador Analógico
#INT_EEPROM	EEIF - Escrita na EEPROM
#INT_EXT	INTF - Interrupção externa RB0/INT
#INT_EXT1	INT1F - Interrupção externa INT1
#INT_EXT2	INT2F - Interrupção externa INT2
#INT_I2C	I2CIF - Interrupção do módulo I2C (PIC 14000)
#INT_LCD	LCDIF - Interrupção do módulo LCD
#INT_LOWVOLT	LVDIF - Detecção de baixa tensão
#INT_PSP	PSPIF - Chegada de dados no módulo PSP
#INT_RB	RBIF - Mudança na porta B (RB4-RB7)
#INT_RC	Mudança na porta C (RC4 - RC7)
#INT_RDA	RCIF - Recepção de dados na USART
#INT_RTCC	T0IF - Estouro de contagem do timer 0
#INT_SSP	SSPIF - Atividade de comunicação SPI ou I2C
#INT_TBE	TXIF - Buffer de transmissão vazio
#INT_TIMER0	T0IF - Estouro de contagem do timer 0
#INT_TIMER1	TMR1IF - Estouro de contagem do timer 1
#INT_TIMER2	TMR2IF - Estouro de contagem do timer 2
#INT_TIMER3	TMR3IF - Estouro de contagem do timer 3

Tabela 10.2

Note que nem todas estas interrupções estão disponíveis num mesmo dispositivo.

Para saber quais as interrupções disponíveis num determinado dispositivo, basta selecionar a opção View > Valid Interrupts no menu principal do compilador PCW.

O compilador gera todo o código necessário para o salvamento/ restauração do estado do processador, além de automaticamente limpar o *flag* relativo à interrupção especificada.

Opcionalmente é possível utilizar a opção NOCLEAR para evitar que o compilador apague o *flag* da interrupção.

Nos PICs da série 18, é possível utilizar ainda a opção FAST para determinar ao compilador que a referida interrupção deve ser setada para alta prioridade.

Exemplo:

```
#int_timer1
void trata_t1()
{
   x++;
}

#int_rb
void trata_rbif()
{

} // fim do tratamento do RBIF
```

#INT_DEFAULT

Especifica uma função padrão para tratamento de interrupção para o caso do evento de uma interrupção sem função de tratamento específica.

Sintaxe:

```
#int_default
```

Exemplo:

```
#int_default
void interrupcao_padrao ()
{
// aqui vai o código para tratamento da interrupção
}
```

#INT_GLOBAL

Esta diretiva pode ser utilizada para substituir a rotina padrão de tratamento de interrupção do compilador.

Sintaxe:

```
#int_global
```

Utilize esta diretiva para evitar que o compilador gere código para tratamento de interrupção.

Caso o programador deseje utilizar esta diretiva, terá de escrever todo o código necessário ao tratamento de interrupções (salvamento do contexto, verificação de qual interrupção ocorreu, código relativo à interrupção, apagamento do flag de interrupção e restauração do contexto).

Note que a função seguinte a esta diretiva será a responsável pelo tratamento de interrupção e será posicionada pelo compilador no endereço 0x0004.

Exemplo:

```
#int_global
void rti()
{
// código de tratamento da interrupção
}
```

__LINE__

O identificador __LINE__ é substituído em tempo de compilação pelo número da linha atual em que a diretiva encontra-se inserida.

Sintaxe:

```
__LINE__
```

#LIST

Inicia ou pára de inserir códigos no arquivo de listagem a partir deste ponto.

Sintaxe:

```
#list
```

#LOCATE

Determina um endereço fixo e específico para uma variável ou identificador C.

Sintaxe:

```
#locate identificador = endereço
```

A diretiva **#locate** funciona como a diretiva **#byte**, com a diferença de que **#locate** cria uma variável global, impedindo que o compilador utilize o endereço para outra variável.

Observe que o *identificador* precisa ser uma variável C previamente definida.

Exemplo:

```
int32 teste;
#locate teste=0x20
```

#NOLIST

Pára de inserir código no arquivo de listagem a partir do ponto em que foi inserida esta diretiva.

Sintaxe:

```
#NOLIST
```

#OPT

Seleciona o nível de otimização para o compilador.

Sintaxe:

```
#OPT n
```

Esta diretiva permite que o programador selecione o nível de otimização a ser adotado pelo compilador.

O 5 é o nível padrão dos compiladores PCB, PCM e PCH. Nos compiladores PCW e PCWH, o nível padrão é o máximo (9).

#ORG

Define o endereço inicial para montagem do programa ou função.

Sintaxe:

```
#org  início, fim
```

ou

```
#org segmento
```

ou

```
#org início, fim {}
```

ou

```
#org início, fim  auto=0
```

Utilize esta diretiva para delimitar os endereços inicial e final para uma função ou constante.

É possível também especificar somente o endereço inicial, no caso de já haver sido definida uma outra diretiva **#org** anteriormente.

Caso a diretiva seja seguida por um par de chaves {}, o segmento de memória será reservado, mas não utilizado pelo compilador.

Exemplo:

```
#org 0x1E00, 0x1FFF
funcao1()
{
// Esta função será localizada no endereço 0x1E00
}

#org 0x1E00
funcao2()
{
// Esta função estará localizada no bloco anterior, após a funcao1
}

#org 0x0800, 0x0820 {}
// O bloco entre 0x0800 e 0x0820 permanecerá vago

#org 0x1C00, 0x1C0F
const char id[]= {"123456789"}
/*
        Localiza esta constante a partir do endereço 0x1C00
        Note que a constante acima será implementada utilizando uma
        tabela de memória, o que fará com que algum código seja adicionado
        antes da tabela propriamente dita
*/
```

__PCB__

Este identificador pode ser utilizado para determinar se o compilador utilizado é o PCB.

Sintaxe:

```
__PCB__
```

Utilize a diretiva **#ifdef** ou **#ifndef** para detectar se a diretiva foi ou não definida.

__PCM__

Este identificador pode ser utilizado para determinar se o compilador utilizado é o PCM.

Sintaxe:

```
__PCM__
```

Utilize a diretiva **#ifdef** ou **#ifndef** para detectar se a diretiva foi ou não definida.

__PCH__

Este identificador pode ser utilizado para determinar se o compilador utilizado é o PCH.

Sintaxe:

```
__PCH__
```

Utilize a diretiva **#ifdef** ou **#ifndef** para detectar se a diretiva foi ou não definida.

#PRAGMA

Utilizada para informar ao compilador que uma diretiva interna vem a seguir.

Sintaxe:

```
#pragma diretiva
```

Repare que esta diretiva não é necessária com o compilador CCS, no entanto, por questão de compatibilidade, a diretiva é aceita e interpretada pelo compilador.

#PRIORITY

Define a ordem de prioridade do atendimento de interrupções.

Sintaxe:

```
#priority int,int...
```

Utilize esta diretiva para definir a ordem de varredura da rotina de tratamento de interrupções. A primeira interrupção (int) definida é a de maior prioridade que a segunda e assim por diante.

Exemplo:

```
#priority timer0,timer1
```

#RESERVE

Reserva um ou mais endereços da memória RAM do dispositivo.

Sintaxe:

```
#reserve endereço
```

ou

```
#reserve endereço, endereço1, ...
```

ou

```
#reserve início, fim
```

Utilize esta diretiva para reservar uma ou mais posições de memória, impendindo o seu uso pelo compilador.

Esta diretiva deve aparecer logo após a diretiva **#device**, ou não produz efeito nenhum.

Exemplo:

```
#device PIC16C74
#reserve 0x60:0x6F // reserva 16 endereços entre 0x60 e 0x6F
```

#ROM

Esta diretiva permite definir valores para serem diretamente armazenados em uma determinada posição de memória.

Sintaxe:

```
#rom endereço = {valor1,valor2,...};
```

A principal finalidade desta diretiva é permitir a programação, no código-fonte, do conteúdo da memória EEPROM interna presente em alguns dispositivos.

Note que esta diretiva não impede que o compilador utilize os endereços especificados.

Nos PICs da série 16, o endereço inicial da EEPROM é 0x2100.

Nos PICs da série 18, o endereço inicial da EEPROM é 0xF00000.

Exemplo:

```
#rom 0x2100={0,1,2,3,4} // programa as cinco primeiras posições da EEPROM
```

#SEPARATE

Informa ao compilador que a próxima função deve ser implementada separadamente.

Sintaxe:

```
#separate
```

Esta diretiva é a contraparte da diretiva **#inline**. **#separate** informa ao compilador para inserir apenas uma cópia da função seguinte no código do programa. Cada referência à função será atendida utilizando o desvio de programa/ chamada de sub-rotina.

Esta diretiva economiza memória ROM do dispositivo, fazendo um uso mais intensivo da pilha.

Exemplo:

```
#separate
swapbyte (int *a, int *b)
{
        int t;
        t=*a;
        *a=*b;
        *b=t;
}
```

__TIME__

Este identificador interno do compilador retorna a hora em que o programa foi compilado.

Sintaxe:

```
__TIME__
```

#TYPE

Modifica os tipos de dados do compilador.

Sintaxe:

```
#type tipo_padrão = tamanho
```

Modifica o tamanho dos tipos de dados utilizados pelo compilador.

O compilador CCS utiliza como padrão o tamanho de 1 bit para o **short**, 8 bits para o tipo **int** e 16 bits para o **long**.

No entanto, para manter a compatibilidade com programas escritos em outros compiladores / plataformas, a diretiva **#type** permite modificar o tamanho de cada um dos tipos básicos de dado do compilador.

Note, entretanto, que os quatro tipos internos do compilador não podem ser redefinidos por esta diretiva: **int1**, **int8**, **int16** e **int32**.

Exemplo:

```
#type short = 8, int = 16, long = 32 // definição tradicional da linguagem
C para microcomputadores
```

#UNDEF

Cancela uma definição feita anteriormente.

Sintaxe:

```
#undef identificador
```

Exemplo:

```
#define teste
#undef teste  //cancela a definição de teste
```

#USE DELAY

Informa ao compilador a velocidade de clock do sistema de destino.

Sintaxe:

```
#use delay (clock = valor)
```

ou

```
#use delay (clock = valor, restart_wdt)
```

Esta diretiva é necessária para a utilização das funções de atraso de tempo (**delay_us()** e **delay_ms()**) e também para a rotina de comunicação serial por *software*.

O *valor* a ser especificado é a velocidade de clock em Hertz (entre 1 e 100000000 Hertz).

Opcionalmente, pode ser utilizada a opção **restart_wdt** para que o compilador insira código de manutenção do *watchdog* nas rotinas de atraso.

Exemplo:

```
#use delay (clock = 1000000)  // informa ao compilador que o clock é 1MHz
#use delay (clock = 10000000, restart_wdt) // clock de 10MHz c/ watchdog
```

#USE STANDARD_IO

Seleciona o modo padrão de entrada e saída.

Sintaxe:

```
#use standard_io (porta)
```

Esta diretiva modifica a forma de tratamento das funções de entrada e saída do compilador.

O parâmetro *porta* especifica qual das portas de E/S será afetada pela diretiva.

No modo padrão de E/S, o compilador irá inserir código de configuração dos registradores TRIS a cada utilização do(s) pino(s) especificado(s).

Se for utilizada uma função de entrada, então o(s) pino(s) é(são) configurado(s) como entrada(s).

Se for utilizada uma função de saída, então o(s) pino(s) é(são) configurado(s) como saída(s).

Nos processadores de 12 bits, esta diretiva irá utilizar um byte de memória RAM para cada porta de E/S utilizada.

Exemplo:

```
#use standard_io (B)
```

#USE FAST_IO

Seleciona o modo rápido de entrada e saída.

Sintaxe:

```
#use fast_io (porta)
```

Esta diretiva modifica a forma de tratamento das funções de entrada e saída do compilador.

A *porta* de entrada e saída atingida pela diretiva dever ser especificada (A,B,C,D,...).

Neste modo, o compilador não irá inserir nenhuma instrução para controle dos registradores TRIS do dispositivo, produzindo um código mais rápido e eficiente que o modo padrão do compilador.

A diretiva permanecerá em utilização até que outra diretiva de controle de E/S seja utilizada.

Exemplo:

```
#use fast_io (B)
```

#USE FIXED_IO

Seleciona o modo fixo de entrada e saída.

Sintaxe:

```
#use fixed_io (porta_outputs = pino, pino,...)
```

Esta diretiva modifica a forma de tratamento das funções de entrada e saída do compilador.

O parâmetro *porta* especifica qual das portas de E/S será afetada pela diretiva e cada *pino* especificado em seguida será configurado como saída.

No modo fixo de E/S, o compilador irá inserir código de configuração dos registradores TRIS a cada utilização do(s) pino(s) especificado(s).

Note que neste modo, o registrador TRIS é programado com a configuração selecionada na diretiva e não pelo tipo de função executado.

A utilização desta diretiva também economiza uma posição de memória RAM que é normalmente ocupada pelo modo padrão de E/S.

Exemplo:

```
#use fixed_io (B)
```

#USE I2C

Habilita o uso da biblioteca interna de comunicação I^2C.

Sintaxe:

```
#use i2c (opções)
```

Opção	Descrição
MASTER	Seta o modo mestre
SLAVE	Seta o modo escravo
SCL = pino	Especifica qual o pino de clock I^2C
SDA = pino	Especifica qual o pino de dados I^2C
ADDRESS = nn	Especifica o endereço do dispositivo em modo escravo
FAST	Utiliza a especificação de alta velocidade I^2C
SLOW	Utiliza a especificação de baixa velocidade I^2C
RESTART_WDT	Reinicia o watchdog enquanto aguarda dados na função I2C_READ
FORCE_HW	Força a utilização do hardware interno (SSP ou MSSP) do PIC

Tabela 10.3

Esta diretiva permite utilizar o protocolo I^2C para comunicação com dispositivos externos, como memórias, conversores A/D, relógios, calendários, etc.

Uma vez definida esta diretiva, é possível utilizar as funções da biblioteca I2C interna do compilador (I2C_START, I2C_STOP, I2C_READ, I2C_WRITE E I2C_POLL).

O protocolo será implementado por software, a não ser que seja especificada a utilização do hardware interno (**force_hw**). Neste caso, certifique-se de que os pinos especificados sejam os mesmos do *hardware* interno do PIC.

O modo escravo deve ser utilizado somente com o módulo SSP interno.

Exemplos:

```
// seleciona o modo mestre com o pino RB0 para dados e o pino RB1 como
saída de clock
#use I2C (master, sda = PIN_B0, scl = PIN_B1)

// seleciona o modo escravo, com pinos RC4 para dados e RC3 para clock,
endereço 0x10 e utilizando o hardware interno do dispositivo
#use I2C (slave, sda = PIN_C4, scl = PIN_C3, address = 0x10, force_hw)
```

#USE RS232

Ativa o suporte à comunicação serial.

Sintaxe:

```
#use rs232 (opções)
```

Opção	Descrição
BAUD = valor	Especifica a velocidade de comunicação serial.
XMIT = pino	Especifica o pino de transmissão de dados.
RCV = pino	Especifica o pino de recepção de dados.
RESTART_WDT	Determina que a função GETC() ressete o watchdog enquanto aguarda a chegada de um caractere.
INVERT	Inverte a polaridade dos pinos de TX/RX. Não pode ser utilizada com o hardware interno.
PARITY = x	Seleciona a paridade (x pode ser: N (sem paridade), E (paridade par) ou O (paridade ímpar)).
BITS = x	Seleciona o número de bits de dados (5 a 9 para o modo por *software* e 8 a 9 para o modo por *hardware*).
FLOAT_HIGH	A saída não vai em nível lógico 1. Utilizado com saídas coletor aberto.
ERRORS	Solicita ao compilador que armazene os erros de recepção na variável RS232_ERRORS, ressetando os flags de erro quando eles ocorrerem.
BRGH1OK	Permite a utilização das velocidades disponíveis com o bit BRGH em 1 em chips que tenham bugs nesta configuração do hardware.
ENABLE = pino	Especifica um pino para atuar como saída de habilitação durante uma transmissão. Utilizado no protocolo RS485.
STREAM = identificador	Associa a interface RS232 a um identificador de *stream* de dados. Os stream de dados são utilizados em algumas funções internas do compilador.

Tabela 10.4

Esta diretiva permite a construção de interfaces seriais por software (ou hardware se ele estiver presente), permitindo a utilização de funções como **printf**, **getc** e **putc**.

Esta diretiva permanece ativa até que outra diretiva #RS232 seja definida.

É necessário utilizar a diretiva #USE DELAY antes dessa diretiva.

Se forem especificados os pinos da interface SCI interna do dispositivo (normalmente pinos RB2 (TX) e RB1 (RX) nos PICs de 18 pinos e RC6 (TX) e RC7 (RX) nos PICs de 28 e 40 pinos) e caso ela esteja presente, a diretiva irá utilizar o hardware de comunicação do PIC.

Se o compilador não puder obter uma velocidade próxima de 3% do valor desejado, será emitida uma mensagem de erro.

A variável RS232_ERRORS possui a seguinte formatação:

- Modo de software: o bit 7 da variável irá armazenar o nono bit de dados e o bit 6 da variável, se setado, indica que não foi possível colocar a saída em alta impedância (modo **float_high**).
- Modo de hardware: quando utilizado o hardware interno, a variável conterá uma cópia do registrador RCSTA, com exceção do bit 0 que é utilizado para indicar erro de paridade.

Exemplo:

```
#use rs232 (baud = 19200, xmit = PIN_A0, rcv = PIN_A3)
```

#ZERO_RAM

Apaga o conteúdo da memória RAM do dispositivo.

Sintaxe:

```
#zero_ram
```

Funções do Compilador

11.1. Matemáticas

ABS()

Retorna o valor absoluto (sem sinal) de um número.

Sintaxe:

```
valor = abs ( x )
```

Em que:

- *x* é do tipo signed int de 8, 16 ou 32 bits, ou float
- *valor* é do mesmo tipo de *x*

Requisito: #include <stdlib.h>

Exemplo:

```
signed int teste, outro;
teste = -5;
outro = abs (teste);  // outro = 5
```

LABS()

Retorna o valor absoluto (sem sinal) de um inteiro longo.

Sintaxe:

```
valor = labs ( x )
```

Em que:

- *x* é do tipo signed long int (16 bits)
- *valor* é do tipo signed long int

Requisito: #include <stdlib.h>

Exemplo:

```
signed long int teste, outro;
teste = -500;
outro = labs (teste);  // outro = 500
```

FABS()

Retorna o valor absoluto (sem sinal) de um número fracionário (ponto flutuante).

Sintaxe:

```
valor = labs (x)
```

Em que:

- *x* é do tipo float
- *valor* é do tipo float

Requisito: #include <math.h>

Exemplo:

```
float teste, outro;
teste = -100.33;
outro = fabs (teste);  // outro = 100.33
```

MODF()

Retorna a parte inteira e a parte fracionária de um número fracionário.

Sintaxe:

```
valor = modf (x , &y)
```

Em que:

- *x* e *y* são do tipo float
- *valor* é do tipo float

Esta função retorna a parte inteira (*valor*) e a parte fracionária de um número (*x*). A parte fracionária é retornada pela passagem por referência da segunda variável (*y*).

Requisito: #include <math.h>

Exemplo:

```
float valor, inteiro, fracao;
valor = -100.33;
inteiro = modf (valor, &fracao);  // outro = -100.00
// fracao = 0.33
```

FMOD()

Retorna a o resto da divisão de dois números fracionários.

Sintaxe:

```
valor = fmod (x , y)
```

Em que:

- *x* e *y* são do tipo float
- *valor* é do tipo float

O resultado será igual a (x - i * y), em que i é um número inteiro. Se y for diferente de zero, o resultado terá o mesmo sinal de x e magnitude menor que y.

Requisito: #include <math.h>

Exemplo:

```
float teste;
teste = fmod (3,2);  // outro = 1
```

CEIL()

Retorna o menor número inteiro que é maior que o valor especificado.

Sintaxe:

```
valor = ceil (x)
```

Em que:

- *x* é do tipo float
- *valor* é do tipo float

Requisito: #include <math.h>

Exemplo:

```
float teste,var1;
teste = 5.79;
var1 = ceil (teste);  // var1 = 6.00
```

FLOOR()

Retorna o menor número inteiro que não é maior que o valor especificado.

Sintaxe:

```
valor = floor (x)
```

Em que:

- *x* é do tipo float
- *valor* é do tipo float

Requisito: #include <math.h>

Exemplo:

```
float teste,var1;
teste = 5.79;
var1 = floor (teste);  // var1 = 5.00
```

SIN(), ASIN(), SINH()

Retorna o valor do seno (sin), arco seno (asin) ou seno hiperbólico (sinh) do valor especificado.

Sintaxe:

```
valor = sin (x)
valor = asin (x)
valor = sinh (x)
```

Em que:

- *x* é do tipo float
- *valor* é do tipo float

Na função seno, x deverá ser um ângulo em radianos (entre -2π e 2π).

Na função arco seno, x deverá ser um valor entre -1.0 e 1.0.

Na função seno hiperbólico, x corresponde a um valor qualquer.

O valor retornado pelo função será:

- um valor entre -1.0 e 1.0 no caso da função seno;
- um ângulo em radianos (entre $-\pi/2$ e $\pi/2$) no caso da função arco seno;
- um valor qualquer no caso do seno hiperbólico.

Requisito: #include <math.h>

Exemplo:

```
float teste, angulo;
angulo = 2 * 3.141596;
teste = sin (angulo);
```

COS(), ACOS(), COSH()

Retorna o valor do coseno (cos), arco coseno (acos) ou coseno hiperbólico (cosh) do valor especificado.

Sintaxe:

```
valor = cos (x)
valor = acos (x)
valor = cosh (x)
```

Em que:

- *x* é do tipo float
- *valor* é do tipo float

Na função seno, x deverá ser um ângulo em radianos (entre -2π e 2π).

Na função arco seno, x deverá ser um valor entre -1.0 e 1.0.

Na função seno hiperbólico, x corresponde a um valor qualquer.

O valor retornado pelo função será:

- um valor entre -1.0 e 1.0 no caso da função seno;
- um ângulo em radianos (entre $-\pi/2$ e $\pi/2$) no caso da função arco seno;
- um valor qualquer no caso do seno hiperbólico.

Requisito: #include <math.h>

Exemplo:

```
float teste, angulo;
angulo = 2 * 3.141596;
teste = cos (angulo);
```

TAN(), ATAN(), ATAN2(), TANH()

Retorna o valor da tangente (tan), arco tangente (atan) ou tangente hiperbólica (tanh) do valor especificado.

Sintaxe:

```
valor = tan (x)
valor = atan (x)
valor = atan2 (x , y)
valor = tanh (x)
```

Em que:

- *x* e *y* são do tipo float
- *valor* é do tipo float

Na função tangente, x deve ser um ângulo em radianos (entre -2π e 2π).

Na função arco tangente, x deve ser um valor entre –1.0 e 1.0.

Na função seno hiperbólico, x corresponde a um valor qualquer.

O valor retornado pela função será:

- o valor da tangente (tan);
- um ângulo em radianos (entre $-\pi/2$ e $\pi/2$) no caso da função arco tangente (atan);
- um ângulo em radianos (entre $-\pi$ e π) correspondente ao arco tangente de y / x (atan2);
- um valor qualquer no caso da tangente hiperbólica (tanh).

Requisito: #include <math.h>

Exemplo:

```
float teste, angulo;
angulo = 2 * 3.141596;
teste = tan (angulo);
```

FREXP()

Decompõe o número em uma mantissa na faixa de 0.5 a 1 e um expoente inteiro tal que o número será igual à mantissa * 2^{EXP}.

Sintaxe:

```
valor = frexp (x , &exp)
```

Em que:

- *x* é do tipo float
- *exp* é do tipo signed int
- *valor* é do tipo float

A mantissa é retornada por *valor* e o expoente é retornado pela variável apontada por *exp*.

Requisito: #include <math.h>

Exemplo:

```
float teste;
signed int expoente;
teste = 10;
teste = frexp (teste, &expoente);
// teste = 0.625
// expoente = 4
```

EXP()

Calcula o valor de e^x.

Sintaxe:

```
valor = exp (x)
```

Em que:

- *x* é do tipo float
- *valor* é do tipo float

O valor de e é aproximadamente igual a 2,718282.

Requisito: #include <math.h>

Exemplo:

```
float teste;
teste = exp (2);
// teste = 7.3891
```

LDEXP()

Multiplica um valor por 2^{exp}.

Sintaxe:

```
valor = ldexp (x , exp)
```

Em que:

- *x* é do tipo float
- *exp* é do tipo signed int
- *valor* é do tipo float

O resultado é igual a x * 2^{exp}.

Requisito: #include <math.h>

Exemplo:

```
float teste;
teste = exp ( 2, 3 );
// teste = 2 * 2³ = 16
```

LOG()

Calcula o valor do logaritmo natural do número especificado.

Sintaxe:

```
valor = log (x)
```

Em que:

- *x* é do tipo float
- *valor* é do tipo float

Requisito: #include <math.h>

Exemplo:

```
float resultado;
resultado = log (10);
// resultado = 2.3026
```

LOG10()

Calcula o valor do logaritmo base 10 do número especificado.

Sintaxe:

```
valor = log10 (x)
```

Em que:

- *x* é do tipo float
- *valor* é do tipo float

Requisito: #include <math.h>

Exemplo:

```
float resultado;
resultado = log10 (10);
// resultado = 1.0000
```

POW()

Calcula o valor de x^y.

Sintaxe:

```
valor = pow (x , y)
```

Em que:

- *x* e *y* são do tipo float
- *valor* é do tipo float

Requisito: #include <math.h>

Exemplo:

```
float resultado;
int potencia = 3;
resultado = pow ( 3 , potencia );
// resultado = 27
```

SQRT()

Calcula o valor de x^y.

Sintaxe:

```
valor = sqrt (x)
```

Em que:

- *x* é do tipo float
- *valor* é do tipo float

Requisito: #include <math.h>

Exemplo:

```
float resultado, valor;
valor = 25;
resultado = sqrt (valor );
// resultado = 5
```

11.2. Manipulação de Caracteres

ATOI(), ATOL(), ATOI32(),ATOF()

Converte uma string em um valor inteiro de 8, 16 ou 32 bits, ou em um float.

Sintaxe:

```
valor_8 = atoi (x)
valor_16 = atol (x)
valor_32 = atoi32 (x)
valor_float = atof (x)
```

Em que:

- *x* é uma string
- *valor_8* é do tipo inteiro
- *valor_16* é do tipo long int
- *valor_32* é do tipo int32
- *valor_float* é do tipo float

Requisito: #include <stdlib.h>

Exemplo:

```
char teste[]={"1200"};
long int valor;
valor = atol(teste);
```

TOLOWER(), TOUPPER()

Converte um caractere no seu correspondente minúsculo (tolower) ou maiúsculo (toupper).

Sintaxe:

```
maiusculo = toupper (x)
minusculo = tolower (x)
```

Em que:

- *x* é um caractere
- *maiusculo e minusculo* são do tipo caractere

Requisito: nenhum

Exemplo:

```
char teste[]={"1200"};
long int valor;
valor = atol(teste);
```

ISALNUM(), ISALPHA(), ISDIGIT(), ISLOWER(), ISSPACE(), ISUPPER(), ISXDIGIT(), ISCNTRL(), ISGRAPH(), ISPRINT(), ISPUNCT()

Testa se o caractere pertence a um determinado grupo.

Sintaxe:

```
valor = isalnum ( x )
valor = isalpha ( x )
valor = isdigit ( x )
valor = islower ( x )
valor = isspace ( x )
valor = isupper ( x )
valor = isxdigit ( x )
valor = iscntrl ( x )
valor = isgraph ( x )
valor = isprint ( x )
valor = ispunct ( x )
```

Em que:

- *x* é um caractere
- *valor* é do tipo booleano (int1)

As funções retornam 1 (verdadeiro) se o caractere pertencer ao grupo especificado ou 0 (falso) caso negativo.

As funções verificam se o caractere é:

- isalnum: uma letra ('a' a 'z', 'A' a 'Z') ou número ('0' a '9');
- isalpha: uma letra ('a' a 'z', 'A' a 'Z');
- isdigit: um número ('0' a '9');
- islower: uma letra minúscula ('a' a 'z');
- isupper: uma letra maiúscula ('A' a 'Z');
- isspace: um espaço (32 decimal);
- isxdigit: um caractere hexadecimal ('0' a '9', 'A' a 'F' ou 'a' a 'f');
- iscntrl: um caractere de controle (abaixo de 32 decimal exclusive);
- isgraph: um caractere gráfico (acima de 32 decimal exclusive);
- isprint: um caractere imprimível (acima de 32 decimal inclusive);
- ispunct: um caractere maior que 32 decimal e que não é letra ou número.

Requisito: #include <ctype.h>

Exemplo:

```
char teste;
teste = 'a';
if (islower(teste)) teste=toupper(teste);
```

ISAMOUNG()

Verifica se um caractere pertence a uma string.

Sintaxe:

```
result = isamoung ( x , string )
```

Em que:

- *x* é um caractere
- *string* é uma constante de caracteres
- *result* é do tipo short int (booleano)

Se o caractere pertencer ao conjunto de caracteres fornecido, o resultado é 1 (verdadeiro); caso contrário, o resultado é 0 (falso).

Requisito: #include <ctype.h>

Exemplo:

```
char teste;
teste = 'a';
if (isamoung(teste,"abcdefgh")) printf("OK\r\n");
```

STRLEN()

Retorna o número de caracteres na string.

Sintaxe:

```
result = strlen ( string )
```

Em que:

- *string* é uma constante de caracteres
- *result* é do tipo int (8 bits)

Observe que a referência de final de string desta função é o caractere nulo!

O caractere nulo não é contado para efeito do número de caracteres da string.

Requisito: #include <string.h>

Exemplo:

```
char teste[]={"Compilador C"};
int tam;
tam = strlen (teste);  // tam = 12
```

STRCPY()

Copia o conteúdo de uma string para outra.

Sintaxe:

```
strcpy ( s1 , s2 )
```

Em que:

- s1 e s2 são strings de caracteres
- A função strcpy copia o conteúdo da string ou constante de caracteres s2 para a string s1

Requisito: #include <string.h>

Exemplo:

```
char teste[15];
strcpy ( teste , "testando ...");
```

STRNCPY()

Copia o conteúdo de uma string para outra.

Sintaxe:

```
strcpy ( s1 , s2 )
ptr = strncpy ( s1 , s2 , n )
```

Em que:

- *s1* e *s2* são strings de caracteres
- *n* é o número de caracteres a ser copiado
- *ptr* é um ponteiro para a string s1

A função strncpy copia n caracteres da string ou constante de caracteres s2 para a string s1. Esta função retorna um ponteiro para a string s1.

Se a string s2 for menor que o número n de caracteres a ser copiado, o número de caracteres faltantes será preenchido com nulos em s1.

Se s2 tiver mais que n caracteres, são copiados n caracteres. Neste caso, a string s1 não será terminada em nulo.

Requisito: #include <string.h>

Exemplo:

```
char teste[15], outra[15];
strcpy ( teste , "testando ...");
strncpy ( outra, teste, 5); // outra="testa"
```

STRCMP(), STRNCMP(), STRICMP()

Compara duas strings e retorna um valor informando se uma é maior, igual ou menor que a outra.

Sintaxe:

```
resultado = strcmp (s1, s2)
resultado = strncmp (s1, s2, n)
resultado = stricmp (s1, s2)
```

Em que:

- *s1* e *s2* são strings de caracteres
- *n* é o número de caracteres a ser comparado
- *resultado* é um signed int com o resultado da comparação

A função strcmp efetua a comparação lexicográfica entre duas strings. O resultado da comparação será:

- negativo: se a string s1 for menor que s2;
- 0: se a string s1 for igual a s2;
- positivo: se a string s1 for maior que s2.

Na função strncmp, a comparação ocorre como anteriormente, mas são comparados apenas os n primeiros caracteres das strings. Se uma das strings for menor que o número n de caracteres a serem comparados, a função terminará ao atingir o caractere nulo da menor string.

Na função stricmp, a comparação é feita como na função strcmp, com a diferença de que os caracteres maiúsculos/minúsculos são tratados como iguais.

Requisito: #include <string.h>

Exemplo:

```
char teste[15], outra[15];
strcpy ( teste , "testando ...");
strncpy ( outra, teste, 5);
```

STRCAT()

Concatena uma string dentro de outra.

Sintaxe:

```
ptr = strcat ( s1 , s2 )
```

Em que:

- *s1* e *s2* são strings de caracteres
- *ptr* é um ponteiro para s1

Esta função copia o conteúdo da string s2 no final da string s1. O terminador nulo de s1 é sobreposto pelo primeiro caractere de s2. A função retorna um ponteiro para a string s1. Observe que a string s1 deve ter tamanho suficiente para armazenar as duas strings.

Requisito: #include <string.h>

Exemplo:

```
char teste[20]={"primeiro "}, outra[]={"segunda"};
strcat ( teste , outra );
```

STRSTR()

Localiza a primeira ocorrência de uma string dentro de outra.

Sintaxe:

```
ptr = strstr ( s1 , s2 )
```

Em que:

- *s1* e *s2* são strings de caracteres
- *ptr* é um ponteiro para s1

Esta função retorna um ponteiro para a primeira ocorrência da string s2 dentro da string s1. Se não for encontrada nenhuma ocorrência, o ponteiro retornado será nulo.

Requisito: #include <string.h>

Exemplo:

```
char teste[20]={"testando..."};
printf ("%s", strstr ( teste , "tes" ));
// imprime "tando..."
```

STRCHR(), STRRCHR()

Localiza a primeira ocorrência de um caractere em uma string.

Sintaxe:

```
ptr = strchr ( s1 , letra )
ptr = strrchr ( s1 , letra )
```

Em que:

- *s1* é uma string de caracteres
- *letra* é um dado char ou inteiro
- *ptr* é um ponteiro para s1

A função strchr localiza a primeira ocorrência do caractere especificado na string s1. Caso o caractere seja localizado, a função retorna um ponteiro para a primeira ocorrência do caractere na string s1; caso não seja encontrado, a função retorna um ponteiro nulo.

A função strrchr realiza a mesma operação descrita anterirormente, apenas iniciando a busca pelo final da string s1.

Requisito: #include <string.h>

Exemplo:

```
char teste[20]={"primeiro"};
char *outra;
outra = strchr ( teste , 'i' );
printf ("%s", outra);
// imprime "meiro"
outra = strrchr ( teste , 'i' );
printf ("%s", outra);
// imprime "ro"
```

STRTOK()

Retorna um ponteiro para a próxima ocorrência de uma palavra em uma string.

Sintaxe:

```
ptr = strtok ( s1 , s2 )
```

Em que:

- *s1* é uma string de caracteres
- *s2* é uma constante de caracteres
- *ptr* é um ponteiro para s1

A função strtok localiza a próxima ocorrência da constante s2 na constante s1.

Os caracteres que formam s2 são os delimitadores da palavra a ser localizada.

A string s1 é modificada pela função!

Requisito: #include <string.h>

Exemplo:

```
char teste[20]={"2testando"};
outra = strtok ( teste , 'te' );
printf ("%s", outra); // imprime "2"
```

STRSPN(), STRCSPN()

Conta o número de caracteres presentes em uma string e que aparecem também em outra.

Sintaxe:

```
resultado = strspn ( s1 , s2 )
resultado = strcspn ( s1 , s2 )
```

Em que:

- *s1* e *s2* são strings de caracteres
- *resultado* é um inteiro de 8 bits

A função strspn retorna o comprimento de uma substring formada pelos caracteres iniciais da string s1 que aparecem também na string s2. A função irá retornar quando atingir o final da string s1 ou encontrar caracteres não presentes em s2.

A função strspn retorna o comprimento de uma substring formada pelos caracteres iniciais da string s1 que não aparecem na string s2. A função irá retornar quando atingir o final da string s1 ou encontrar algum dos caracteres presentes em s2.

Em ambas, os caracteres não precisam estar na mesma ordem.

Requisito: #include <string.h>

Exemplo:

```
char teste[20]={"primeiro"}, outro[]={"primo"};
printf ("%d\r\n",strspn(teste,outro));
// imprime o número 4
```

STRPBRK()

Localiza um caractere de uma string dentro de outra.

Sintaxe:

```
ptr = strpbrk ( s1 , s2 )
```

Em que:

- *s1* e *s2* são strings de caracteres
- *ptr* é um ponteiro para s1

A função strpbrk procura na string s1 a primeira ocorrência de quaisquer dos caracteres presentes em s2.

Caso a função encontre algum dos caracteres, a posição dele é retornada pelo ponteiro.

Caso não seja encontrado nenhum dos caracteres, é retornado um ponteiro nulo.

Requisito: #include <string.h>

Exemplo:

```
char teste[20]={"primeiro"};
char *outra;
outra = strrchr ( teste , 'abcdef' );
printf ("%s", outra); // imprime "eiro"
```

STRLWR()

Converte os caracteres de uma string em minúsculos.

Sintaxe:

```
ptr = strpbrk ( s1 )
```

Em que:

- *s1* é uma string de caracteres
- *ptr* é um ponteiro para s1

A função strlwr converte os caracteres da string s1 em minúsculos. É retornado um ponteiro para a string.

Lembre-se que a string s1 é alterada pela função !

Requisito: #include <string.h>

Exemplo:

```
char teste[20]={"PRImeiro"};
char *outra;
outra = strlwr ( teste );
printf ("%s", outra); // imprime "primeiro"
printf ("%s", teste); // imprime "primeiro"
```

SPRINTF()

Imprime uma string ou constante de caracteres em outra string.

Sintaxe:

```
sprintf ( s1 , s2 , variáveis )
```

Em que:

- *s1* é uma string de caracteres
- *s2* é uma constante de caracteres
- *variáveis* é uma lista de variáveis separadas por vírgulas

Esta função trabalha praticamente da mesma forma que a função printf, com a diferença de que sprintf redireciona a saída de impressão para uma string s1.

Os códigos de formatação válidos para printf são também válidos para sprintf.

Esta função não retorna valores.

Requisito: #include <string.h>

Exemplo:

```
char x1[10];
float a = 5.89024;
sprintf (x1,"%2.2f",a);
// a string x1 será igual a 5.89
```

11.3. Memória

MEMSET()

Inicializa uma ou mais posições de memória com um determinado valor.

Sintaxe:

```
memset( destino , valor , n )
```

Em que:

- *destino* é um ponteiro para uma posição de memória
- *valor* é o valor a ser armazenado na memória
- *n* é o número de posições de memória a serem preenchidas

Esta função preenche a memória RAM a partir do endereço especificado pelo ponteiro *destino*, por *n* posições, com o *valor* especificado.

Exemplo:

```
memset (matriz1, 0, sizeof(matriz1));
memset (&estrutura, 0x10, sizeof(estrutura));
```

MEMCPY()

Copia n bytes de posição de memória para outra.

Sintaxe:

```
memcpy( destino , origem , n )
```

Em que:

- *destino* é um ponteiro para a posição de memória que receberá os dados
- *origem* é um ponteiro para a posição de memória da qual os dados serão lidos
- *n* é o número de posições de memória a serem copiadas

Esta função copia *n* bytes a partir do local especificado pelo ponteiro *origem*, para o endereço de memória especificado pelo ponteiro *destino*.

Exemplo:

```
memcpy (&estrutura1, &estrutura2, sizeof(estrutura1));
memcpy (&estrutura1, &dado, 1);
```

OFFSETOF(), OFFSETOFBIT()

Retornam o deslocamento em bytes ou bits de um elemento de uma estrutura.

Sintaxe:

```
valor = offsetof ( estr , campo )
valor = offsetofbit ( estr , campo )
```

Em que:

- *estr* é o nome de uma estrutura
- *campo* é o nome de um elemento da estrutura
- *valor* é um inteiro de 8 bits

Estas funções retornam o deslocamento em bytes (offsetof) ou em bits (offsetofbit) de um campo *campo* de uma estrutura *estr*.

Exemplo:

```
struct tempo
{
int hora, minuto, segundo;
int mes: 4;
boolean horario_verao;
}
int desloc;
desloc = offsetof (tempo, minuto);
// desloc = 1, ou seja o segundo byte da estrutura
desloc = offsetofbit (tempo, segundo);
// desloc = 16, ou seja, 16 bits após o início da
// estrutura
desloc = offsetof (tempo, horario_verao);
// desloc = 3, ou seja, o campo está localizado no
// quarto byte da estrutura
```

11.4. Atraso

DELAY_CYCLES()

Aguarda n ciclos de máquina.

Sintaxe:

```
delay_cycles ( n )
```

Em que:

- *n* é uma constante inteira de 8 bits

Esta função retarda a execução do programa por *n* ciclos de máquina.

Exemplo:

```
delay_cycles ( 5 ); // atrasa 5 ciclos de máquina
```

DELAY_US()

Aguarda n microssegundos.

Sintaxe:

```
delay_us ( n )
```

Em que:

- *n* é uma variável inteira de 8 bits ou uma constante de 16 bits

Esta função retarda a execução do programa por *n* microssegundos.

Exemplo:

```
delay_us ( 1 ); // aguarda 1 us.
```

DELAY_MS()

Aguarda n milissegundos.

Sintaxe:

```
delay_ms ( n )
```

Em que:

- *n* é uma variável inteira de 8 bits ou uma constante de 16 bits

Esta função retarda a execução do programa por *n* milissegundos.

Exemplo:

```
delay_ms ( 1 ); // aguarda 1 ms.
```

11.5. Manipulação de Bit / Byte

SHIFT_RIGHT()

Insere um bit em uma variável, matriz ou estrutura, deslocando à direita os demais bits dela.

Sintaxe:

```
bit = shift_right ( e , n , v )
```

Em que:

- *e* é um ponteiro para a variável
- *n* é um número inteiro correspondendo ao tamanho em bytes da variável que será deslocada
- *v* é uma variável ou constante booleana representando o valor do bit a ser inserido (0 ou 1)
- *bit* é uma variável booleana

O bit *v* é inserido no bit 7 da parte mais significativa da variável apontada por *e*, sendo os bits da variável deslocados à direita. O valor do bit LSB (bit 0 do endereço mais baixo da memória RAM) que for deslocado para fora da variável é retornado pela função.

Exemplo:

```
long int teste = 0x2500;
shift_right( &teste, 2, 0); // teste =0x1280
```

SHIFT_LEFT()

Insere um bit em uma variável, matriz ou estrutura, deslocando à esquerda os demais bits dela.

Sintaxe:

```
bit = shift_right ( e , n , v )
```

Em que:

- *e* é um ponteiro para a variável
- *n* é um número inteiro correspondendo ao tamanho em bytes da variável que será deslocada
- *v* é uma variável ou constante booleana representando o valor do bit a ser inserido (0 ou 1)
- *bit* é uma variável booleana

O bit *v* é inserido no bit 0 da parte menos significativa da variável apontada por *e*, sendo os bits da variável deslocados à esquerda. O valor do bit MSB (bit 7 do endereço mais alto da memória RAM) que for deslocado para fora da variável é retornado pela função.

Exemplo:

```
long int teste = 0x2500;
shift_left( &teste, 2, 0); // teste =0x4A00
```

ROTATE_RIGHT()

Rotaciona à direita uma variável, matriz ou estrutura de dados.

Sintaxe:

```
rotate_right ( e , n )
```

Em que:

- *e* é um ponteiro para a variável
- *n* é um número inteiro correspondendo ao tamanho em bytes da variável que será deslocada

Um valor 0 é inserido no bit 7 da parte mais significativa da variável apontada por *e*, sendo os bits da variável deslocados à direita. O valor do bit LSB (bit 0 do endereço mais baixo da memória RAM) que for deslocado para fora é descartado.

Exemplo:

```
long int teste = 0x2500;
rotate_right( &teste, 2 ); // teste =0x1280
```

ROTATE_LEFT()

Rotaciona à esquerda uma variável, matriz ou estrutura de dados.

Sintaxe:

```
rotate_left ( e , n )
```

Em que:

- *e* é um ponteiro para a variável
- *n* é um número inteiro correspondendo ao tamanho em bytes da variável que será deslocada

Um valor 0 é inserido no bit 0 da parte menos significativa da variável apontada por *e*, sendo os bits da variável deslocados à esquerda. O valor do bit MSB (bit 7 do endereço mais alto da memória RAM) que for deslocado para fora é descartado.

Exemplo:

```
long int teste = 0x2500;
rotate_left( &teste, 2 ); // teste =0x4A00
```

BIT_CLEAR()

Apaga um bit de uma variável.

Sintaxe:

```
bit_clear ( var , bit )
```

Em que:

- *var* é uma variável inteira de 8, 16 ou 32 bits
- *bit* é um número de 0 a 31 especificando qual dos bits da variável será apagado

Exemplo:

```
long int teste = 0x2502;
bit_clear( teste, 1 ); // teste =0x2500
```

BIT_SET()

Seta um bit de uma variável.

Sintaxe:

```
bit_clear ( var , bit )
```

Em que:

- *var* é uma variável inteira de 8, 16 ou 32 bits
- *bit* é um número de 0 a 31 especificando qual dos bits da variável será setado

Exemplo:

```
long int teste = 0x2502;
bit_set( teste, 0 ); // teste =0x2503
```

BIT_TEST()

Testa um bit de uma variável.

Sintaxe:

```
res = bit_test ( var , bit )
```

Em que:

- *var* é uma variável inteira de 8, 16 ou 32 bits
- *bit* é um número de 0 a 31 especificando qual dos bits da variável será apagado
- *res* é um valor booleano indicando o estado do bit testado

Esta função testa um determinado bit (*bit*) de uma variável (*var*). O estado do bit é retornado pela função.

Exemplo:

```
long int teste = 0x2502;
x = boolean;
x = bit_test( teste, 1 ); // x = 1
```

SWAP()

Troca os conteúdos da parte alta pela parte baixa de uma variável de 8 bits.

Sintaxe:

```
swap ( var )
```

Em que:

- *var* é uma variável inteira de 8 bits

Esta função inverte o conteúdo de uma variável de 8 bits: passando o nibble superior para a parte inferior e vice-versa.

Nenhum valor é retornado pela função.

Exemplo:

```
int teste = 0x25;
swap( teste ); // teste = 0x52
```

MAKE8()

Lê um dos bytes que compõem uma variável de 16 ou 32 bits.

Sintaxe:

```
res = make8 ( var , desloc )
```

Em que:

- *var* é uma variável inteira de 16 ou 32 bits
- *desloc* é um valor inteiro que especifica qual dos bytes da variável deve ser lido

A função make8 efetua a leitura de byte especificado por *desloc* da variável *var*. *Desloc* pode ser um valor entre 0 e 3, sendo 0 o LSB e 3 o MSB de uma variável de 32 bits.

Exemplo:

```
long int teste = 0x2502;
int x;
x = make8 ( teste , 1 ) // x = 0x25
```

MAKE16()

Transforma dois valores de 8 bits em um valor de 16 bits.

Sintaxe:

```
res = make16 ( xx , yy )
```

Em que:

- *res* é uma variável inteira de 16 bits
- *xx* e *yy* são duas variáveis de 8 bits

O valor de 16 bits será igual a *xxyy*.

Exemplo:

```
long int teste;
int a = 0x25,b = 0x02;
teste = make16 ( a , b ) // teste = 0x2502
```

MAKE32()

Transforma um ou mais (até 4) valores de 8 ou 16 bits.

Sintaxe:

```
res = make32 (a, b, c, d )
```

Em que:

- *res* é uma variável inteira de 32 bits
- *a*, *b*, *c* e *d* são variáveis de 8 ou 16 bits

Esta função cria uma variável inteira de 32 bits a partir de um ou mais valores de 8 ou 16 bits.

O parâmetro *a* é o MSB do valor de 32 bits. Os parâmetros *b*, *c* e *d* são opcionais.

Se os valores fornecidos não totalizarem os 32 bits, serão inseridos zeros no MSB até que a variável final possua os 32 bits necessários.

Exemplo:

```
int32 teste;
int a = 0x25,b = 0x02;
teste = make32 ( a , b ) // teste = 0x00002502
```

11.6. Entrada / Saída

OUTPUT_LOW()

Coloca o pino especificado do microcontrolador em nível 0.

Sintaxe:

```
output_low ( pino )
```

Em que:

- *pino* é um identificador do pino do MCU

O identificador *pino* é definido em todos os arquivos de cabeçalho dos microcontroladores e especifica na realidade um endereço formado do seguinte modo:

```
(endereço_e/s) * 8 + (número do bit)
```

Ou seja: supondo o pino 1 da porta B: 0x06 * 8 + 1 = 49.

Lembre-se que o funcionamento das funções de E/S é alterado pelas diretivas de configuração de E/S.

Exemplo:

```
output_low (pin_b1);  // coloca o pino RB1 em 0
```

OUTPUT_HIGH()

Coloca o pino especificado do microcontrolador em nível 1.

Sintaxe:

```
output_low ( pino )
```

Em que:

- *pino* é um identificador do pino do MCU

O identificador *pino* é definido em todos os arquivos de cabeçalho dos microcontroladores e especifica na realidade um endereço formado do seguinte modo:

```
(endereço_e/s) * 8 + (número do bit)
```

Ou seja: supondo o pino 0 da porta A: 0x05 * 8 + 0 = 40.

Lembre-se que o funcionamento das funções de E/S é alterado pelas diretivas de configuração de E/S.

Exemplo:

```
output_high (pin_a0);  // coloca o pino RA0 em 1
```

OUTPUT_FLOAT()

Coloca o pino especificado do microcontrolador em estado de alta impedância (entrada).

Sintaxe:

```
output_float ( pino )
```

Em que:

- *pino* é um identificador do pino do MCU

O identificador *pino* é definido em todos os arquivos de cabeçalho dos microcontroladores e especifica na realidade um endereço formado do seguinte modo:

```
(endereço_e/s) * 8 + (número do bit)
```

Ou seja: supondo o pino 2 da porta B: 0x06 * 8 + 2 = 50.

Exemplo:

```
output_float (pin_b2);
// configura o pino RB2 como entrada
```

OUTPUT_BIT()

Coloca o pino especificado do microcontrolador em um determinado nível lógico.

Sintaxe:

```
output_bit ( pino )
```

Em que:

- *pino* é um identificador do pino do MCU

O identificador *pino* é definido em todos os arquivos de cabeçalho dos microcontroladores e especifica na realidade um endereço formado do seguinte modo:

```
(endereço_e/s) * 8 + (número do bit)
```

Ou seja: supondo o pino 0 da porta A: 0x05 * 8 + 0 = 40.

Lembre-se que o funcionamento das funções de E/S é alterado pelas diretivas de configuração de E/S.

Exemplo:

```
output_bit (pin_a0 , 0);  // coloca o pino RA0 em 0
```

INPUT()

Lê o estado lógico de um pino do microcontrolador.

Sintaxe:

```
res = input ( pino )
```

Em que:

- *pino* é um identificador do pino do MCU
- *res* é o estado lógico lido do pino

O identificador *pino* é definido em todos os arquivos de cabeçalho dos microcontroladores e especifica na realidade um endereço formado do seguinte modo:

```
(endereço_e/s) * 8 + (número do bit)
```

Ou seja: supondo o pino 0 da porta A: 0x05 * 8 + 0 = 40.

Lembre-se que o funcionamento das funções de E/S é alterado pelas diretivas de configuração de E/S.

Exemplo:

```
short int x;
x = input (pin_a0);  // lé o estado do pino RA0
```

OUTPUT_X()

Escreve um byte completo em uma determinada porta do microcontrolador.

Sintaxe:

```
Output_A ( valor )
output_B ( valor )
output_C ( valor )
output_D ( valor )
output_E ( valor )
```

Em que:

- *valor* é uma variável ou constante inteira de 8 bits

Observação: Nem todos os microcontroladores possuem todas as portas implementadas.

Exemplo:

```
output_b (0x25);
// Escreve o valor 0x25 na porta B
```

INPUT_X()

Lê um byte completo de uma determinada porta do microcontrolador.

Sintaxe:

```
valor = input_A ()
valor = input_B ()
valor = input_C ()
valor = input_D ()
valor = input_E ()
```

Em que:

- *valor* é uma variável inteira de 8 bits

Observação: Nem todos os microcontroladores possuem todas as portas implementadas.

Exemplo:

```
int x;
x = input_b();  // Lê o estado da porta B
```

PORT_B_PULLUPS()

Ativa / desativa os pullups internos da porta B.

Sintaxe:

```
port_b_pullups( estado )
```

Em que:

- *estado* é um valor booleano que determina o estado dos pullups (true - ligado, false - desligado)

Exemplo:

```
port_b_pullups ( true); // liga pullups
```

SET_TRIS_X()

Configura a direção dos pinos de uma porta do microcontrolador.

Sintaxe:

```
set_tris_A ( valor )
set_tris_B ( valor )
set_tris_C ( valor )
set_tris_D ( valor )
set_tris_E ( valor )
```

Em que:

- *valor* é uma variável ou constante inteira de 8 bits

Esta diretiva pode ser utilizada para configurar a direção de funcionamento dos pinos de uma determinada porta de E/S.

Note que o uso desta diretiva é desnecessário com as diretivas #USE STANDARD_IO e #USE FIXED_IO, já que nestes modos de E/S o compilador configura automaticamente a direção dos pinos.

Observação: Nem todos os microcontroladores possuem todas as portas implementadas.

Exemplo:

```
set_tris_b (0x0f);
// Configura os pinos RB0 a RB3 como entradas e RB4 a RB7 como saídas
```

11.7. Analógicas

SETUP_COMPARATOR()

Configura o funcionamento do módulo comparador interno.

Sintaxe:

```
setup_comparator ( valor )
```

Em que:

- *valor* é uma variável ou constante inteira de 8 bits

Os valores possíveis para configuração dos comparadores são normalmente:

```
A0_A3_A1_A2
A0_A2_A1_A2
```

```
NC_NC_A1_A2
NC_NC_NC_NC
A0_VR_A1_VR
A3_VR_A2_VR
A0_A2_A1_A2_OUT_ON_A3_A4
A3_A2_A1_A2
```

Maiores detalhes sobre a configuração do módulo comparador podem ser obtidos no datasheet do fabricante, ou no livro Microcontroladores PIC: Técnicas Avançadas.

Exemplo:

```
setup_comparator ( nc_nc_nc_nc );
// desliga comparadores
```

SETUP_VREF()

Configura a referência interna de tensão.

Sintaxe:

```
setup_vref ( modo | valor )
```

Em que:

- *modo* é uma constante que especifica o modo de funcionamento da referência
- *valor* é uma constante inteira de 4 bits (0 a 15)

Os valores possíveis para configuração da referência interna são:

- FALSE: referência desligada
- VREF_LOW: VDD = valor / 24
- VREF_HIGH: VDD = valor / 32 + VDD / 4

É ainda possível utilizar a opção VREF_A2 para ativar a saída externa da Referência de tensão. Esta opção pode ser adicionada às opções anteriores por meio de uma operação OR (|).

Valor	Tensão de saída	
	VREF_LOW	VREF_HIGH
0	0,00 V	1,25 V
1	0,21 V	1,41 V
2	0,42 V	1,56 V
3	0,63 V	1,72 V
4	0,83 V	1,88 V

Valor	Tensão de saída	
	VREF_LOW	VREF_HIGH
5	1,04 V	2,03 V
6	1,25 V	2,19 V
7	1,46 V	2,34 V
8	1,67 V	2,50 V
9	1,88 V	2,66 V
10	2,08 V	2,81 V
11	2,29 V	2,97 V
12	2,50 V	3,13 V
13	2,71 V	3,28 V
14	2,92 V	3,44 V
15	3,13 V	3,59 V

O valor determina o nível de tensão da referência conforme a tabela seguinte (para VDD = 5V):

Maiores detalhes sobre a configuração do módulo comparador podem ser obtidos no datasheet do fabricante, ou no livro Microcontroladores PIC : Técnicas Avançadas.

Exemplo:

```
setup_vref ( VREF_LOW | VREF_A2 | 2);
// configura o comparador para saída externa de 0,42 V
```

SETUP_ADC()

Configura o conversor AD interno.

Sintaxe:

```
setup_ADC ( opcões )
```

Em que:

- *opções* é uma variável ou constante inteira de 8 bits

Esta diretiva pode ser utilizada para configurar o conversor AD interno. As opções variam de acordo com cada PIC e podem ser encontradas no arquivo de cabeçalho de cada dispositivo.

As mais comuns são:

```
ADC_OFF
ADC_CLOCK_DIV_2
ADC_CLOCK_DIV_4
```

```
ADC_CLOCK_DIV_8
ADC_CLOCK_DIV_16
ADC_CLOCK_DIV_32
ADC_CLOCK_DIV_64
ADC_CLOCK_INTERNAL
```

Exemplo:

```
setup_adc(adc_off);  // desliga o conversor A/D
```

SETUP_ADC_PORTS()

Configura as entradas analógicas do conversor A/D interno.

Sintaxe:

```
setup_ADC_ports ( opcões )
```

Em que:

- *opções* é uma variável ou constante inteira de 8 bits

Esta diretiva pode ser utilizada para configurar os pinos de entrada do conversor AD interno. As opções variam de acordo com cada PIC e podem ser encontradas no arquivo de cabeçalho de cada dispositivo.

As opções de configuração válidas são:

Modo	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	Vref+	Vref-
ALL_ANALOG	A	A	A	A	A	A	A	A	VDD	VSS
ANALOG_RA3_REF	A	A	A	A	Vref+	A	A	A	AN3	VSS
A_ANALOG	D	D	D	A	A	A	A	A	VDD	VSS
A_ANALOG_RA3_REF	D	D	D	A	Vref+	A	A	A	AN3	VSS
RA0_RA1_RA3_ANALOG	D	D	D	D	A	D	A	A	VDD	VSS
RA0_RA1_ANALOG_RA3_REF	D	D	D	D	Vref+	D	A	A	AN3	VSS
NO_ANALOGS	D	D	D	D	D	D	D	D	-	-
ANALOG_RA3_RA2_REF	A	A	A	A	Vref+	Vref-	A	A	AN3	AN2
ANALOG_NOT_RE1_RE2	D	D	A	A	A	A	A	A	VDD	VSS
ANALOG_NOT_RE1_RE2_REF_RA3	D	D	A	A	Vref+	A	A	A	AN3	VSS
ANALOG_NOT_RE1_RE2_REF_RA3_RA2	D	D	A	A	Vref+	Vref-	A	A	AN3	AN2
A_ANALOG_RA3_RA2_REF	D	D	D	A	Vref+	Vref-	A	A	AN3	AN2
RA0_RA1_ANALOG_RA3_RA2_REF	D	D	D	D	Vref+	Vref-	A	A	AN3	AN2
RA0_ANALOG	D	D	D	D	D	D	D	A	VDD	VSS
RA0_ANALOG_RA3_RA2_REF	D	D	D	D	Vref+	Vref-	D	A	AN3	AN2

Note que somente os PICs de 40 pinos com A/D interno possuem todas as entradas analógicas da tabela anterior.

Exemplo:

```
setup_adc_ports(no_analogs);  // todos os pinos digitais
```

SET_ADC_CHANNEL()

Seleciona um canal de entrada para o módulo AD interno.

Sintaxe:

```
set_adc_channel ( valor )
```

Em que:

- *valor* é uma variável ou constante inteira de 8 bits

Lembre-se de após a seleção de um canal, aguardar pelo menos 10µs antes de efetuar a conversão A/D, de forma a garantir que o capacitor interno de amostragem possa ser carregado com a tensão de entrada.

Nem todos os microcontroladores possuem todas as portas implementadas.

Exemplo:

```
long int valor;
set_adc_channel (1);
delay_us(10);
valor = read_adc();
```

READ_ADC()

Efetua uma conversão A/D.

Sintaxe:

```
valor = read_adc ()
```

Em que:

- *valor* é uma variável inteira de 8 ou 16 bits

Esta função inicia um ciclo de conversão A/D e somente retorna a execução após o término do ciclo de conversão.

Lembre-se de após a seleção de um canal, aguardar pelo menos T_{AQ} (20µs para impedâncias de entrada de 10 kohms) antes de efetuar a conver-

são A/D, de forma a garantir que o capacitor interno de amostragem possa ser carregado com a tensão de entrada.

Exemplo:

```
long int valor;
set_adc_channel (1);
delay_us(10);
valor = read_adc();  // inicia e aguarda a conversão A/D
```

11.8. Manipulação de Timers

SETUP_TIMER_0(), SETUP_RTCC()

Configura o timer 0.

Sintaxe:

```
setup_timer_0( modo )
setup_rtcc ( modo )
```

Em que:

- *modo* é uma variável ou constante inteira de 8 bits

O *modo* pode ser uma das seguintes constantes:

- RTCC_INTERNAL: Timer 0 com clock interno;
- RTCC_EXT_L_TO_H: Timer 0 com clock externo sensível à borda de subida;
- RTCC_EXT_H_TO_L: Timer 0 com clock externo sensível à borda de descida;
- RTCC_DIV_2: Prescaler dividindo por 2;
- RTCC_DIV_4: Prescaler dividindo por 4;
- RTCC_DIV_8: Prescaler dividindo por 8;
- RTCC_DIV_16: Prescaler dividindo por 16;
- RTCC_DIV_32: Prescaler dividindo por 32;
- RTCC_DIV_64: Prescaler dividindo por 64;
- RTCC_DIV_128: Prescaler dividindo por 128;
- RTCC_DIV_256: Prescaler dividindo por 256;
- RTCC_OFF: Timer 0 desligado (somente PIC18 e 18F);
- RTCC_8_BIT: Timer 0 em modo 8 bits (somente PIC18 e 18F).

Note que as constantes que configuram o prescaler podem ser unidas a constantes de configuração do timer 0 por um operador OR (|).

Exemplo:

```
setup_timer_0 ( RTCC_INTERNAL | RTCC_DIV_2 );
```

SETUP_TIMER_1()

Configura o timer 1.

Sintaxe:

```
setup_timer_1( modo )
```

Em que:

- *modo* é uma variável ou constante inteira de 8 bits

O *modo* pode ser uma ou mais constantes dos seguintes grupos:

- T1_DISABLED: desliga o timer 1;
- T1_INTERNAL: com clock interno;
- T1_EXTERNAL: clock externo assíncrono;
- T1_EXTERNAL_SYNC: clock externo síncrono;
- T1_CLOCK_OUT: ativa saída externa de clock;
- T1_DIV_BY_1: prescaler do timer 1 dividindo por 1;
- T1_DIV_BY_2: prescaler do timer 1 dividindo por 2;
- T1_DIV_BY_4: prescaler do timer 1 dividindo por 4;
- T1_DIV_BY_8: prescaler do timer 1 dividindo por 8.

As constantes anterior podem ser juntadas por meio de operação OR (|).

Exemplo:

```
setup_timer_1 ( T1_INTERNAL | T1_DIV_BY_4 );
```

SETUP_TIMER_2()

Configura o timer 2.

Sintaxe:

```
setup_timer_2( modo , per , ps )
```

Em que:

- *modo* é uma variável ou constante inteira de 8 bits
- *per* é uma variável inteira de 8 bits
- *ps* é um número entre 1 e 16

O *modo* pode ser uma ou mais constantes dos seguintes grupos:

- T2_DISABLED: desliga o timer 2;
- T2_DIV_BY_1: timer 2 ligado com prescaler dividindo por 1;
- T2_DIV_BY_4: timer 2 ligado com prescaler dividindo por 4;
- T2_DIV_BY_16: timer 2 ligado com prescaler dividindo por 16.

O parâmetro *per* é utilizado para especificar um período de contagem para o timer 2. A contagem do temporizador inicia em 0 e cada vez que ele chegar a uma contagem igual ao valor de *per*, o temporizador será reiniciado.

O parâmetro *ps* é utilizado para especificar um valor para o pós-divisor do timer 2. O número programado em *ps* será igual ao número de reinícios do timer 2, necessários para gerar uma interrupção (TMR2IF). Isto significa que se for programado um *ps* = 2, a cada dois reinícios de contagem do timer 2, será gerada uma interrupção.

Exemplo:

```
setup_timer_2 ( T1_DIV_BY_1 , 100 , 1 );
```

SETUP_TIMER_3()

Configura o timer 3.

Sintaxe:

```
setup_timer_3( modo )
```

Em que:

- *modo* é uma variável ou constante inteira de 8 bits

O *modo* pode ser uma ou mais constantes dos seguintes grupos:

- T3_DISABLED: desliga o timer 3;
- T3_INTERNAL: seleciona o clock interno;
- T3_EXTERNAL: seleciona o clock externo assíncrono;
- T3_EXTERNAL_SYNC: seleciona o clock externo síncrono;
- T3_DIV_BY_1: prescaler dividindo por 1;

- T3_DIV_BY_2: prescaler dividindo por 2;
- T3_DIV_BY_4: prescaler dividindo por 4;
- T3_DIV_BY_8: prescaler dividindo por 8.

É possível agrupar constantes dos dois grupos anteriores por meio da operação OR (|).

O timer 3 somente está disponível nos PICs da linha 18 e 18F.

Exemplo:

```
setup_timer_3 ( T3_INTERNAL | T3_DIV_BY_2 );
```

SETUP_COUNTERS()

Configura o timer 0 e watchdog nos dispositivos de 12 bits.

Sintaxe:

```
setup_counters( modo , ps );
```

Em que:

- *modo* é uma variável ou constante inteira de 8 bits
- *ps* é uma variável ou constante inteira de 8 bits

O *modo* pode ser uma das seguintes constantes:

- RTCC_INTERNAL: Timer 0 com clock interno;
- RTCC_EXT_L_TO_H: Timer 0 com clock externo sensível à borda de subida;
- RTCC_EXT_H_TO_L: Timer 0 com clock externo sensível à borda de descida.

O valor de *ps* pode ser um dos seguintes:

- RTCC_DIV_2: Prescaler ligado ao timer 0 e dividindo por 2;
- RTCC_DIV_4: Prescaler ligado ao timer 0 e dividindo por 4;
- RTCC_DIV_8: Prescaler ligado ao timer 0 e dividindo por 8;
- RTCC_DIV_16: Prescaler ligado ao timer 0 e dividindo por 16;
- RTCC_DIV_32: Prescaler ligado ao timer 0 e dividindo por 32;
- RTCC_DIV_64: Prescaler ligado ao timer 0 e dividindo por 64;
- RTCC_DIV_128: Prescaler ligado ao timer 0 e dividindo por 128;
- RTCC_DIV_256: Prescaler ligado ao timer 0 e dividindo por 256;

- WDT_18MS: Prescaler ligado ao watchdog, timeout de 18ms;
- WDT_36MS: Prescaler ligado ao watchdog, timeout de 36ms;
- WDT_72MS: Prescaler ligado ao watchdog, timeout de 72ms;
- WDT_144MS: Prescaler ligado ao watchdog, timeout de 144ms;
- WDT_288MS: Prescaler ligado ao watchdog, timeout de 288ms;
- WDT_576MS: Prescaler ligado ao watchdog, timeout de 576ms;
- WDT_1152MS: Prescaler ligado ao watchdog, timeout de 1,152 s;
- WDT_2304MS: Prescaler ligado ao watchdog, timeout de 2,304 s.

Por meio do parâmetro *ps*, é possível configurar o prescaler para trabalhar com o timer 0 ou com o watchdog.

Exemplo:

```
setup_counters( RTCC_INTERNAL , RTCC_DIV_2 );
```

SET_TIMERX()

Modifica o conteúdo de um timer interno.

Sintaxe:

```
set_rtcc ( valor )
set_timer0 ( valor )
set_timer1 ( valor )
set_timer2 ( valor )
set_timer3 ( valor )
```

Em que:

- *valor* é uma variável ou constante inteira de 8 ou 16 bits

O valor a ser escrito depende do tipo de temporizador e do modelo do PIC em questão.

Nos modelos com núcleo de 14 bits, somente estão disponíveis no máximo três timers:

- Timer 0 ou RTCC, de 8 bits;
- Timer 1, de 16 bits;
- Timer 2, de 8 bits.

Nos modelos com núcleo de 16 bits, existem até quatro timers:

- Timer 0, de 8 ou 16 bits;
- Timer 1, de 16 bits;

- Timer 2, de 8 bits;
- Timer 3, de 16 bits.

Exemplo:

```
set_timer0 ( 50);
// inicializa o timer 0 com 50 decimal
```

GET_TIMERX()

Lê o conteúdo de um timer interno.

Sintaxe:

```
valor = get_rtcc()
valor = get_timer0()
valor = get_timer1()
valor = get_timer2()
valor = get_timer3()
```

Em que:

- *valor* é uma variável ou constante inteira de 8 ou 16 bits

O valor retornado pela função depende do tipo de temporizador e do modelo do PIC em questão.

Nos modelos com núcleo de 14 bits, somente estão disponíveis no máximo três timers:

- Timer 0 ou RTCC, de 8 bits;
- Timer 1, de 16 bits;
- Timer 2, de 8 bits.

Nos modelos com núcleo de 16 bits, existem até quatro timers:

- Timer 0, de 8 ou 16 bits;
- Timer 1, de 16 bits;
- Timer 2, de 8 bits;
- Timer 3, de 16 bits.

Exemplo:

```
int x;
x = get_timer0 ();
// lê o timer 0
```

SETUP_WDT()

Configura o funcionamento do watchdog.

Sintaxe:

```
setup_wdt ( modo )
```

Em que:

- *modo* é uma variável ou constante inteira de 8 bits

O parâmetro modo deve indicar uma das seguintes configurações:

- WDT_18MS: Prescaler ligado ao watchdog, timeout de 18ms;
- WDT_36MS: Prescaler ligado ao watchdog, timeout de 36ms;
- WDT_72MS: Prescaler ligado ao watchdog, timeout de 72ms;
- WDT_144MS: Prescaler ligado ao watchdog, timeout de 144ms;
- WDT_288MS: Prescaler ligado ao watchdog, timeout de 288ms;
- WDT_576MS: Prescaler ligado ao watchdog, timeout de 576ms;
- WDT_1152MS: Prescaler ligado ao watchdog, timeout de 1,152 s;
- WDT_2304MS: Prescaler ligado ao watchdog, timeout de 2,304 s.

Nas famílias PIC18 e 18F temos ainda:

- WDT_ON: liga o watchdog;
- WDT_OFF: desliga o watchdog.

Exemplo:

```
setup_wdt ( WDT_576MS );
```

RESTART_WDT()

Reinicia a contagem do watchdog.

Sintaxe:

```
restart_wdt ()
```

Esta função deve ser utilizada periodicamente para ressetar ou reiniciar a contagem do watchdog, impedindo que ele provoque um reset na CPU.

Exemplo:

```
restart_wdt ();
```

11.9. Comparação / Captura / PWM

SETUP_CCPX()

Configura o funcionamento do watchdog.

Sintaxe:

```
setup_ccp1 ( modo )
setup_ccp2 ( modo )
```

Em que:

- *modo* é uma variável ou constante inteira de 8 bits

Configura o módulo de captura, comparação e geração de sinais PWM. As constantes definidas para a configuração do modo são:

- CCP_OFF: módulo CCP desligado;
- CCP_CAPTURE_FE: captura na borda de descida do sinal;
- CCP_CAPTURE_RE: captura na borda de subida do sinal;
- CCP_CAPTURE_DIV_4: captura a cada 4ª borda de subida do sinal;
- CCP_CAPTURE_DIV_16: captura a cada 16ª borda de subida do sinal;
- CCP_COMPARE_SET_ON_MATCH: modo de comparação, seta pino CCPx;
- CCP_COMPARE_CLR_ON_MATCH: modo de comparação, apaga pino CCPx;
- CCP_COMPARE_INT: modo de comparação com geração de interrupção;
- CCP_COMPARE_RESET_TIMER: modo de comparação com reset do timer 1ou timer 3;
- CCP_PWM: configura o modo de geração de sinal PWM.

O compilador define ainda as variáveis inteiras longas CCP_1 e CCP_2 para o acesso aos registradores de comparação do microcontrolador.

Exemplo:

```
setup_ccp1 ( CCP_CAPTURE_RE );
```

SET_PWMX_DUTY()

Configura o ciclo ativo do módulo CCP no modo PWM.

Sintaxe:

```
set_pwm1_duty ( valor )
set_pwm2_duty ( valor )
```

Em que:

- *valor* é uma variável ou constante inteira de 8 ou 16 bits

O valor especificado como argumento da função pode ser de 8 ou 16 bits:

- Se 8 bits, o valor é armazenado diretamente no registrador CCPR1L ou CCPR2L, dependendo do módulo CCP em questão;
- Se 16 bits, o valor é rotacionado de forma a ser armazenado nos registradores CCPR1L ou CCPR2L e CCP1CON ou CCP2CON, dependendo do módulo CCP em questão.

Note que a utilização de valores de 8 bits é mais eficiente que valores de 16 bits.

Exemplo:

```
set_pwm1_duty ( 100 );
```

11.10. Manipulação da EEPROM Interna

READ_EEPROM()

Lê um byte de um endereço especificado da EEPROM interna.

Sintaxe:

```
valor = read_eeprom ( end )
```

Em que:

- *valor* é um inteiro de 8 bits
- *end* é uma variável ou constante inteira de 8 bits

Esta função efetua a leitura de um determinado endereço (*end*) da memória EEPROM interna. O conteúdo da posição de memória especificado é retornado pela função.

Exemplo:

```
int x;
x = read_eeprom ( 1 );
// lê o endereço 1 da memória EEPROM
```

WRITE_EEPROM()

Escreve um valor em um determinado endereço da memória EEPROM interna.

Sintaxe:

```
write_eeprom ( end , valor )
```

Em que:

- *valor* é uma variável ou constante de 8 bits
- *end* é uma variável ou constante inteira de 8 bits

Esta função efetua a escrita de um valor (*valor*) em um determinado endereço (*end*) da memória EEPROM interna.

Exemplo:

```
write_eeprom ( 1, 5);
// escreve o valor 5 no endereço 1 da
// memória EEPROM interna
```

READ_PROGRAM_EEPROM()

Lê um byte de um endereço especificado da memória de programa.

Sintaxe:

```
valor = read_program_eeprom ( end )
```

Em que:

- *valor* é um inteiro de 8 ou 16 bits
- *end* é uma variável ou constante inteira de 16 ou 32 bits

Esta função efetua a leitura de um determinado endereço (*end*) da memória de programa do dispositivo.

Para dispositivos com núcleo de 12 e 14 bits, o endereço (*end*) deve ser um valor de 16 bits. O valor retornado é um inteiro de 8 bits.

Para dispositivos com núcleo de 16 bits, o endereço (*end*) especificado deve ser um valor 32 bits e o valor retornado pela função é de 16 bits.

Observe que até a presente data, somente os PIC 16F87x e PIC 18 podem efetuar a leitura da memória de programa.

Exemplo:

```
int x;
x = read_program_eeprom ( 1 );
// lê o endereço 1 da memória de programa
```

WRITE_PROGRAM_EEPROM()

Escreve um valor na memória de programa do dispositivo.

Sintaxe:

```
write_program_eeprom ( end , valor )
```

Em que:

- *valor* é um inteiro de 8 ou 16 bits
- *end* é uma variável ou constante inteira de 16 ou 32 bits

Esta função efetua a escrita de um valor em um determinado endereço (*end*) da memória de programa do dispositivo.

Para dispositivos com núcleo de 12 e 14 bits, o endereço (*end*) deve ser um valor de 16 bits. O *valor* a ser escrito é um inteiro de 8 bits.

Para dispositivos com núcleo de 16 bits, o endereço (*end*) especificado deve ser um valor 32 bits e o *valor* a ser escrito é de 16 bits.

Observe que até a presente data, somente os PIC 16F87x e PIC 18 podem efetuar a leitura da memória de programa.

Exemplo:

```
write_program_eeprom ( 10, 0x0050 )
// escreve o valor 0x0050 no endereço 10 da memória
// de programa
```

READ_CALIBRATION()

Lê um valor de calibração do PIC 14000.

Sintaxe:

```
valor = read_calibration ( n )
```

Em que:

- *valor* é um inteiro de 8 bits
- *n* é uma variável ou constante inteira de 8 bits

Esta função efetua a leitura da posição *n* da memória de calibração do PIC 14000.

Exemplo:

```
int x;
x = read_calibration ( 0 );
```

11.11. Controle do Processador

SLEEP()

Coloca o PIC em modo SLEEP.

Sintaxe:

```
sleep ()
```

O compilador insere automaticamente uma instrução SLEEP ao final do bloco de comandos da função **main()**.

Exemplo:

```
sleep();
```

RESET_CPU()

Reinicia o processador.

Sintaxe:

```
reset_cpu ()
```

Esta função atua como um reset a quente.

Nos PICs com núcleo de 12 ou 14 bits, o compilador simplesmente desvia o programa para o vetor de reset (endereço 0 da memória RAM). Nem os GPRs nem os SFRs são alterados por esse reset (com exceção dos registradores PCL e PCLATH).

Nos PICs com núcleo de 16 bits, o compilador insere uma instrução assembly RESET, provocando um reset do processador. Os registradores são alterados para o seu valor padrão de reset.

Exemplo:

```
reset_cpu();
```

RESTART_CAUSE()

Retorna o motivo de reinicialização do processador.

Sintaxe:

```
valor = reset_cpu ()
```

Em que:

- *valor* é um inteiro de 8 bits

Esta função retorna um valor indicativo do motivo de reinicialização do processador. Existem algumas constantes predefinidas em cada arquivo de cabeçalho de dispositivo. Alguns exemplos:

- WDT_FROM_SLEEP: o watchdog reiniciou a CPU durante o modo SLEEP;
- WDT_TIMEOUT: o watchdog reiniciou a CPU durante o funcionamento normal;
- MCLR_FROM_SLEEP: reset de hardware durante o modo SLEEP;
- NORMAL_POWER_UP: reset normal após ligar o chip;
- BROWNOUT_RESTART: reset após queda de tensão de alimentação.

Exemplo:

```
int x;
x = restart_cause();
```

ENABLE_INTERRUPTS()

Habilita uma interrupção.

Sintaxe:

```
enable_interrupts ( valor )
```

Em que:

- *valor* é uma constante inteira de 8 bits

Esta função pode ser utilizada para ativar um ou mais tipos de interrupção do processador.

O valor utilizado como argumento deve ser uma das constantes:

- GLOBAL - GIE: Habilitação global de interrupções;
- INT_AD - ADIF: Conversão A/D completa;

- INT_ADOF - ADOF: Overflow da conversão A/D;
- INT_BUSCOL - BCLIF: Colisão de barramento (I2C);
- INT_BUTTON: Pushbutton;
- INT_CCP1 - CCP1IF: Módulo CCP1;
- INT_CCP2 - CCP2IF: Módulo CCP2;
- INT_COMP - CMIF: Comparador Analógico;
- INT_EEPROM - EEIF: Escrita na EEPROM;
- INT_EXT - INTF: Interrupção externa RB0/INT;
- INT_EXT1 - INT1F: Interrupção externa INT1;
- INT_EXT2 - INT2F: Interrupção externa INT2;
- INT_I2C - I2CIF: Interrupção do módulo I2C (PIC 14000);
- INT_LCD - LCDIF: Interrupção do módulo LCD;
- INT_LOWVOLT - LVDIF: Detecção de baixa tensão;
- INT_PSP - PSPIF: Chegada de dados no módulo PSP;
- INT_RB - RBIF: Mudança na porta B (RB4-RB7);
- INT_RC: Mudança na porta C (RC4 - RC7);
- INT_RDA - RCIF: Recepção de dados na USART;
- INT_RTCC - T0IF: Estouro de contagem do timer 0;
- INT_SSP - SSPIF: Atividade de comunicação SPI ou I2C;
- INT_TBE - TXIF: Buffer de transmissão vazio;
- INT_TIMER0 - T0IF: Estouro de contagem do timer 0;
- INT_TIMER1 - TMR1IF: Estouro de contagem do timer 1;
- INT_TIMER2 - TMR2IF: Estouro de contagem do timer 2;
- INT_TIMER3 - TMR3IF: Estouro de contagem do timer 3.

Note que nem todas as interrupções estarão disponíveis em um mesmo dispositivo. Para maiores informações, consulte o menu View > Valid Interrupts, os arquivos de cabeçalho de cada dispositivo ou o datasheet do dispositivo.

É possível agrupar mais de uma constante em uma mesma função, utilizando o operador OR (|), desde que estejam localizadas no mesmo registrador de controle.

Exemplo:

```
enable_interrupts ( GLOBAL | INT_TIMER0 | INT_RB );
// habilita o bit GIE, T0IE e RBIE
```

DISABLE_INTERRUPTS()

Desabilita uma interrupção.

Sintaxe:

```
disable_interrupts ( valor )
```

Em que:

- *valor* é uma constante inteira de 8 bits

Esta função pode ser utilizada para desativar um ou mais tipos de interrupção do processador.

As constantes e observações feitas para a função enable_interrupts são válidas também para esta função.

Exemplo:

```
disable_interrupts ( GLOBAL | INT_TIMER0 | INT_RB );
// desabilita o bit GIE, T0IE e RBIE
```

EXT_INT_EDGE()

Seleciona a borda de sensibilidade de interrupção externa.

Sintaxe:

```
ext_int_edge ( sel , borda )
```

Em que:

- *sel* é uma constante inteira entre 0 e 2
- *borda* é uma constante inteira

O argumento *sel* seleciona uma das fontes externas de interrupção disponíveis:

- 0: INT0;
- 1: INT1;
- 2: INT2.

Lembre-se que as duas últimas somente estão disponíveis nos PICs da série 18.

O argumento *borda* seleciona uma das opções de ativação da interrupção externa:

- H_TO_L: ativação da interrupção na borda de descida do sinal;
- L_TO_H: ativação da interrupção na borda de subida do sinal.

Exemplo:

```
ext_int_edge ( 0, L_TO_H );
// seleciona a borda de subida da interrupção externa 0
```

READ_BANK()

Efetua a leitura de uma posição da memória RAM (registradores GPR).

Sintaxe:

```
valor = read_bank ( banco , desloc )
```

Em que:

- *valor* é um valor inteiro de 8 bits
- *banco* é uma constante inteira de 8 bits
- *desloc* é uma constante ou variável de 8 bits

A função read_bank pode ser utilizada para efetuar a leitura de uma das posições de memória RAM disponível ao usuário. Ela efetua uma leitura indireta da memória por meio dos registradores FSR e INDF.

Nos PICs das séries 12 e 14, o valor do argumento *banco* é utilizado para modificar o bit IRP no registrador STATUS, de acordo com o banco selecionado. O argumento *desloc* é carregado no registrador FSR e o conteúdo do registrador INDF é retornado pela função.

Nos PICs da série 18 e 18F, o valor do argumento *banco* é carregado em um dos registradores FSRxH, o argumento *desloc* é carregado no registrador FSRxL e o conteúdo do registrador INDFx é retornado pela função.

O valor efetivamente carregado no registrador FSR ou FSRxL pode ser diferente do argumento *desloc*, uma vez que o compilador irá inserir código para que o registrador apontado seja sempre um GPR.

Exemplo:

```
int x;
x = read_bank ( 0, 0);
// lê a primeira posição de memória RAM do banco 0
// (endereço 0x20 em um PIC 16F877)
```

WRITE_BANK()

Escreve em uma posição da memória RAM (registradores GPR).

Sintaxe:

```
write_bank ( banco , desloc , valor )
```

Em que:

- *banco* é uma constante inteira de 8 bits
- *desloc* é uma constante ou variável de 8 bits
- *valor* é uma variável ou constante inteira de 8 bits

A função write_bank pode ser utilizada para efetuar a escrita em uma das posições de memória RAM disponível ao usuário.

Nos PICs das séries 12 e 14, o valor do argumento *banco* é utilizado para modificar o bit IRP no registrador STATUS, de acordo com o banco selecionado. O argumento *desloc* é carregado no registrador FSR e o conteúdo do argumento *valor* é carregado no registrador INDF, o que provoca a escrita na posição especificada da memória RAM.

Nos PICs da série 18 e 18F, o valor do argumento *banco* é carregado em um dos registradores FSRxH e o argumento *desloc* é carregado no registrador FSRxL, o argumento *valor* é carregado no registrador INDFx, fazendo com que seja escrita a posição especificada da memória RAM.

O valor efetivamente carregado no registrador FSR ou FSRxL pode ser diferente do argumento *desloc*, uma vez que o compilador irá inserir código para que o registrador apontado seja sempre um GPR.

Exemplo:

```
write_bank ( 0, 0, 10 );
// escreve o valor 10 decimal na primeira posição de RAM
// do banco 0 (endereço 0x20)
```

LABEL_ADDRESS()

Retorna o endereço de memória onde está localizado o rótulo.

Sintaxe:

```
end = label_address ( rotulo )
```

Em que:

- *end* é um valor de 16 ou 32 bits
- *rotulo* é um identificador para um rótulo +bel)

Nos PICs com núcleo de 12 ou 14 bits, o endereço retornado é um valor de 16 bits.

Nos PICs com núcleo de 16 bits, o endereço retornado é um valor de 32 bits.

Exemplo:

```
long int endereco;
int x;
teste:
x = 2;
endereco = label_address ( teste );
```

GOTO_ADDRESS()

Desvia o programa para um determinado endereço.

Sintaxe:

```
goto_address ( end )
```

Em que:

- *end* é um valor de 16 ou 32 bits

Esta função efetua o desvio do programa para o endereço *end* especificado.

Exemplo:

```
long int endereco;
int x;
teste:
x = 2;
endereco = label_address ( teste );
x++;
goto_address ( endereco );
```

11.12. Porta Paralela Escrava

SETUP_PSP()

Ativa / desativa a porta paralela escrava.

Sintaxe:

```
setup_psp ( PSP_ENABLED )
setup_psp ( PSP_DISABLED )
```

Esta função pode ser utilizada para ativar (PSP_ENABLED) ou desativar (PSP_DISABLED) a porta paralela escrava (PSP) nos dispositivos que a possuam.

Antes de utilizar a PSP, é necessário configurar adequadamente o respectivo registrador TRIS (normalmente TRISE), o que pode ser feito com a função SET_TRIS_X.

O conteúdo da PSP pode ser lido ou modificado pela variável *psp_data*, definida pelo compilador.

Exemplo:

```
setup_psp (PSP_ENABLED);
```

PSP_INPUT_FULL()

Verifica se um novo dado está presente na entrada da PSP.

Sintaxe:

```
res = psp_input_full ()
```

Em que:

- *res* é um valor booleano

Esta função retorna 1 (true) quando um novo dado estiver presente na entrada da PSP (bit IBF do registrador TRISE ou PSPCON em 1). Caso contrário, retorna 0 (false).

Exemplo:

```
int x;
if (psp_input_full()) x = psp_data;
```

PSP_OUTPUT_FULL()

Verifica se o valor armazenado no PSP já foi lido por um dispositivo externo.

Sintaxe:

```
res = psp_output_full ()
```

Em que:

- *res* é um valor booleano

Esta função retorna 1 (true) caso já tenha sido efetuada a leitura do dado presente na saída da PSP pelo dispositivo externo (bit OBF do registrador TRISE ou PSPCON em 1). Caso contrário, retorna 0 (false).

Exemplo:

```
if (psp_output_full()) psp_data = x [++y];
```

PSP_OVERFLOW()

Verifica se houve sobreescrita de dados na entrada da PSP.

Sintaxe:

```
res = psp_overflow ()
```

Em que:

- *res* é um valor booleano

Esta função retorna 1 (true) quando um novo dado estiver presente na entrada da PSP, sem que o anterior tenha sido lido pelo processador (bit IBOV do registrador TRISE ou PSPCON em 1). Caso contrário, retorna 0 (false).

Esta função automaticamente apaga o bit IBOV.

Exemplo:

```
if (psp_overflow()) erro();
```

11.13. Comunicação Serial Assíncrona

GETC(), GETCH(), GETCHAR()

Aguarda a chegada de um caractere pela porta serial padrão e retorna o seu valor.

Sintaxe:

```
valor = getc ()
valor = getch ()
valor = getchar()
```

Em que:

- *valor* é um valor inteiro de 8 bits

A função GETC aguarda a chegada de um caractere pela linha serial.

Caso seja utilizada a USART interna, esta função pode retornar imediatamente caso existam caracteres no buffer de recepção (que armazena um máximo de três caracteres).

Observe que no caso da não-utilização da USART interna, somente pode ocorrer a recepção de um caractere durante a execução da função.

O caractere recebido é retornado pela função.

Requisito: #use rs232

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2)
int x;
x = getc();
// aguarda a recepção de um caractere pela interface
// serial
```

FGETC()

Aguarda a chegada de um caractere pela stream especificada e retorna o seu valor.

Sintaxe:

```
valor = fgetc ( stream )
```

Em que:

- *valor* é um valor inteiro de 8 bits
- *stream* é o nome da stream utilizada

A função FGETC aguarda a chegada de um caractere por meio da *stream* especificada.

Observe que esta função age da mesma forma que GETC, com a diferença de que aquela opera com o dispositivo padrão de entrada (definido em C como STDIN), ao passo que esta permite a utilização de outros dispositivos de entrada, redirecionados pela stream.

O caractere recebido é retornado pela função.

Requisito: #use rs232

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2, stream = teste)
int x;
x = fgetc( teste );
// aguarda a recepção de um caractere pela interface
// serial teste
```

PUTC(), PUTCHAR()

Escreve um caractere na saída serial padrão.

Sintaxe:

```
putc ( dado )
putchar ( dado )
```

Em que:

- *dado* é um valor inteiro de 8 bits

Esta função pode ser utilizada para escrever um dado ou caractere de 8 bits (*dado*) no dispositivo padrão de saída da linguagem C (STDOUT), que no presente caso é aquele definido pela diretiva #use rs232 padrão.

Requisito: #use rs232

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2)
char x;
x = 'a';
putc ( x );
// envia o caractere 'a' pela saída serial padrão
```

FPUTC()

Escreve um caractere na saída serial especificada pela stream.

Sintaxe:

```
putc ( dado , stream )
```

Em que:

- *valor* é um valor inteiro de 8 bits
- *stream* é o nome da stream utilizada

Esta função pode ser utilizada para escrever um dado ou caractere de 8 bits (*dado*) no dispositivo de saída especificado pela *stream*.

Requisito: #use rs232

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2, stream = teste)
char x;
x = 'a';
fputc ( x , teste );
// envia o caractere 'a' pela saída serial teste
```

GETS()

Lê uma string pela entrada serial padrão.

Sintaxe:

```
gets ( dado )
```

Em que:

- *dado* é uma string de caracteres

Lê uma string de caracteres pela entrada serial padrão (STDIN).

Esta função realiza sucessivas chamadas à função GETC até que o caractere recebido seja um retorno de carro (13 decimal). A string retornada é terminada em nulo.

Requisito: #use rs232

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2)
char x[25];
gets( x );
printf ("%s\r\n", x );
// aguarda um string e imprime de volta a mesma
```

GET_STRING()

Lê uma string com determinado número máximo de caracteres pela entrada serial padrão.

Sintaxe:

```
get_string ( dado , comp )
```

Em que:

- *dado* é uma string de caracteres
- *comp* é um inteiro

Lê uma string de caracteres pela entrada serial padrão (STDIN).

Esta função realiza sucessivas chamadas à função GETC até que o caractere recebido seja um retorno de carro (13 decimal), ou o número máximo de caracteres seja atingido.

A string retornada é terminada em nulo.

Requisito: #use rs232 , include <input.c>

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2)
char x[6];
get_string( x , 5 );
printf ("%s\r\n", x );
// aguarda um string e imprime de volta a mesma
```

FGETS()

Lê uma string com determinado número máximo de caracteres pela stream especificada.

Sintaxe:

```
fgets ( dado , stream )
```

Em que:

- *dado* é uma string de caracteres
- *stream* é o nome de uma stream

Lê uma string de caracteres pela *stream* especificada.

Esta função realiza sucessivas chamadas à função GETC até que o caractere recebido seja um retorno de carro (13 decimal), ou o número máximo de caracteres seja atingido.

A string retornada é terminada em nulo.

Requisito: #use rs232

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2, stream = teste)
char x[25];
fgets( x , teste );
printf ("%s\r\n", x );
// aguarda a chegada de uma string pela stream teste e
// imprime-a pela saída serial padrão (STDOUT)
```

PUTS()

Escreve uma string na saída serial padrão.

Sintaxe:

```
puts ( dado )
```

Em que:

- *dado* é uma string de caracteres

Escreve uma string de caracteres pela saída serial padrão (STDOUT).

Esta função realiza sucessivas chamadas à função PUTC até que o último caractere (nulo) seja encontrado.

A função não transmite o caractere nulo, mas insere um retorno de carro (13) e um avanço de linha (10).

Requisito: #use rs232

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2)
puts ("testando...");
// imprime: testando... pela saída serial padrão
```

FPUTS()

Escreve uma string na stream especificada.

Sintaxe:

```
fputs ( dado , stream )
```

Em que:

- *dado* é uma string de caracteres
- *stream* é o nome de uma stream

Escreve uma string de caracteres pela *stream* especificada.

Esta função realiza sucessivas chamadas à função PUTC até que o último caractere (nulo) seja encontrado.

A função não transmite o caractere nulo, mas insere um retorno de carro (13) e um avanço de linha (10).

Requisito: #use rs232

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2, stream = teste)
fputs ("testando...", teste);
// imprime: testando... pela stream especificada
```

PRINTF(), FPRINTF()

Imprime um dado na saída serial padrão (STDOUT), em uma stream de dados ou em uma função especificada.

Sintaxe:

```
printf ( string )
printf ( string , vari )
fprintf ( string , stream )
fprintf ( string , vari , stream )
printf ( func , string )
printf ( func , string , vari )
```

Em que:

- *string* é uma constante de caracteres
- *vari* é uma lista de uma ou mais variáveis
- *func* é o nome de uma função

A função printf pode escrever dados em três destinos diferentes:

- na saída padrão STDOUT (printf);
- na stream especificada (fprintf);
- em uma função especificada (printf(func , ...)).

Esta função pode ser utilizada para escrever outros tipos de dados que não sejam caracteres ou strings. Para isso, o argumento string deve conter os códigos de formatação necessários à impressão correta do tipo de dado desejado.

Os códigos de formatação obedecem ao formato ***%wt***, em que ***w*** é um número opcional para indicar o número de caracteres a serem impressos e ***t*** pode ser uma das seguintes opções:

- c: caractere;
- s: string ou caractere;
- u: inteiro sem sinal;
- x: inteiro em formato hexadecimal com letras minúsculas;
- X: inteiro em formato hexadecimal com letras maiúsculas;
- d: inteiro com sinal;
- e: ponto flutuante em formato exponencial;
- f: ponto flutuante em formato decimal;
- Lx: hexadecimal longo (16 ou 32 bits) com letras minúsculas;
- LX: hexadecimal longo (16 ou 32 bits) com letras maiúsculas;
- Lu: decimal longo (16 ou 32 bits) sem sinal;
- Ld: decimal longo (16 ou 32 bits) com sinal;
- %: símbolo de porcentagem.

O argumento w funciona da seguinte forma:

- Um número simples, indica o número de caracteres a serem impressos. Se o dado a ser impresso tiver menos caracteres que o número especificado, serão adicionados espaços;
- Um número precedido de 0 funciona da mesma forma descrita anteriormente, com a diferença de que em vez de serem adicionados espaços, são adicionados zeros à esquerda do número;

- Um número seguido por um ponto e outro número especifica o formato de impressão de um número fracionário. O primeiro número especifica o número de casas antes do ponto decimal e o segundo número especifica o número de casas depois do ponto decimal. Quando utilizado com o formato string, o primeiro número especifica o número mínimo de caracteres a serem impressos e o segundo número o número máximo de caracteres a serem impressos. Se o número de caracteres da string for maior que o número máximo estipulado, os caracteres excedentes serão truncados.

```
Requisito: #use rs232 (a não ser que seja utilizada uma função como saída)
```

Exemplos:

```
#uses rs232 (baud = 4800, xmit=pin_b3, rcv=pin_b2)
#uses rs232 (baud = 4800, xmit=pin_b1, rcv=pin_b0, stream = teste);
int x = 12;
signed int t = -5;
long int y = 550;
float z = 250.498567;
char a[] = {"teste da funcao printf"};
printf ("testando ...");
// imprime: testando ...
printf ("O valor de x eh: %u\r\n",x);
// imprime: O valor de x eh: 12
printf ("x = %u, y = %lu, t = %d\r\n",x,y,t);
// imprime: x = 12, y = 550, t = -5
printf ("%05lu\r\n", x);
// imprime: 00550
printf ("%5.3f\r\n",z);
// imprime: 250.498
printf ("%5.12s\r\n",a);
// imprime: teste da fun
printf ("x = %u, y = %lu, t = %d\r\n",x,y,t,teste);
// imprime: x = 12, y = 550, t = -5 na stream teste
printf (imp_lcd, "teste");
// envia a palavra "teste" para a função imp_lcd
```

KBHIT()

Verifica se há um caractere sendo recebido pela porta serial padrão (STDIN).

Sintaxe:

```
res = kbhit ()
```

Em que:

- *res* é um valor booleano

Esta função verifica se há um caractere sendo recebido (no caso da interface serial por software) ou um caractere recebido está disponível no buffer de recepção da USART (RCREG).

Note que para não ocorrerem perdas de dados no caso de comunicação por software, esta função deve ser chamada no mínimo a uma taxa de dez vezes a velocidade de comunicação utilizada. Ou seja: para uma velocidade de 1200 Bps, esta função deveria ser chamada 12000 vezes por segundo.

A função retorna 1 (true) caso um caractere esteja disponível para ser lido, ou 0 (false) caso não exista caractere disponível para recepção.

Requisito: #use rs232

Exemplo:

```
boolean erro_timeout;
...
char getc_temporizado();
/*
esta função aguarda 0,5 segundos pela recepção de
um caractere. Caso seja recebido algum nesse período,
ele é retornado pela função. Caso contrário, a
função retorna o valor 0;
*/
{
   long tempo;
   erro_timeout = false;
   while (!kbhit() && (++tempo<50000) ) delay_us (10);
   if (kbhit()) return ( getc() ); else
      {
         erro_timeout = true;
         return (0);
      }
}
```

SET_UART_SPEED()

Altera a velocidade de comunicação da USART interna.

Sintaxe:

```
set_uart_speed ( vel )
```

Em que:

- *vel* é uma constante entre 100 e 115200

Requisito: #use rs232

Exemplo:

```
#uses rs232 (baud = 4800, xmit=pin_c6, rcv=pin_c7)
...
set_uart_speed ( 9600 );
```

PERROR()

Imprime uma mensagem de erro no dispositivo padrão STDERR.

Sintaxe:

```
perror ( string )
```

Em que:

- *string* é uma string de caracteres

Requisito: #use rs232, errno.h

Exemplo:

```
x = sin ( y );
if (errno!=0) perror ("Erro na função seno");
```

ASSERT()

Verifica uma condição e imprime mensagem de erro no dispositivo STDERR.

Sintaxe:

```
assert ( condição )
```

Em que:

- *Condição* é uma expressão relacional qualquer

Esta função gera uma mensagem de erro no caso de a condição especificada ser avaliada como verdadeira.

Essa mensagem é impressa no dispositivo STDERR, que por padrão é a última diretiva #USE RS232 antes da função.

Na mensagem são incluídos o nome do arquivo e o número da linha de código em que a função assert está localizada.

Este tipo de função pode ser útil durante a depuração de um programa. Para desativá-la, basta inserir a diretiva #define NODEBUG no início do programa. Isto fará com que não seja gerado código para nenhuma das ocorrências desta função.

Requisito: #use rs232, assert.h

Exemplo:

```
char teste [20];
int x;
...
assert ( x>= 20);
teste [x] = 'b';
// se o valor de x for maior ou igual a 20, imprime
// mensagem de erro
```

11.14. Comunicação I²C

I2C_START()

Inicia uma condição de START no barramento I²C no modo mestre.

Sintaxe:

```
i2c_start ()
```

Esta função faz com que a linha de clock seja mantida em nível lógico 0 até que seja chamada a função I2C_WRITE ou uma nova chamada à função I2C_START seja feita, caso em que ocorrerá uma condição especial do protocolo I²C chamada de *Repeated Start*.

No protocolo I²C, a condição de START é definida como a transição de SCL de 1 para 0 com SDA = 0.

Requisito: #use i2c

Exemplo:

```
i2c_start();
// inicia a comunicação
i2c_write(0xc1);
// endereça um dispositivo e ativa flag de leitura
conta = 0;
while (conta!=4)
{
   while (!i2c_poll()); // aguarda chegada dos dados
   buffer[conta++] = i2c_read();
   // armazena o byte no buffer
}
i2c_stop(); // finaliza comunicação
```

I2C_STOP()

Inicia uma condição de STOP no barramento I²C no modo mestre.

Sintaxe:

```
i2c_stop ()
```

A condição de STOP do protocolo I²C é definida como a transição da linha SDA de 0 para 1 com a linha SCL em 1.

Requisito: #use i2c

Exemplo:

```
i2c_start();
// inicia a comunicação
```

```
i2c_write(0xc1);
// endereça um dispositivo e ativa flag de leitura
conta = 0;
while (conta!=4)
{
   while (!i2c_poll()); // aguarda chegada dos dados
   buffer[conta++] = i2c_read();
   // armazena o byte no buffer
}
i2c_stop(); // finaliza comunicação
```

I2C_READ()

Efetua a leitura de um byte do barramento I^2C.

Sintaxe:

```
res = i2c_read ()
res = i2c_read ( ack )
```

Em que:

- *res* e *ack* são valores booleanos

Lembre-se de que no modo escravo, é necessário que ao menos um dispositivo conectado ao barramento I^2C esteja configurado como mestre.

Além disso, o sistema pode ficar paralisado à espera da recepção do byte e caso o watchdog esteja habilitado, pode ocorrer um reset provocado por ele. Nestes casos, é interessante utilizar a opção RESTART_WDT junto à diretiva #USE I2C.

O argumento *ack* que pode ser passado à função indicando a condição de confirmação ou não do dado (*acknowledge*): 0 indica a não confirmação e 1 indica a confirmação. Por padrão, a função considera sempre *ack* = 1.

Requisito: #use i2c

Exemplo:

```
i2c_start();
// inicia a comunicação
i2c_write(0xc1);
// endereça um dispositivo e ativa flag de leitura
conta = 0;
while (conta!=4)
{
   while (!i2c_poll()); // aguarda chegada dos dados
   buffer[conta++] = i2c_read();
   // armazena o byte no buffer
}
i2c_stop(); // finaliza comunicação
```

I2C_WRITE()

Escreve um byte no barramento I²C.

Sintaxe:

```
ack = i2c_write ( dado )
```

Em que:

- *ack* é um valor booleano
- *dado* é um valor inteiro de 8 bits

A escrita de um dado no barramento I²C depende do modo de funcionamento do dispositivo: no modo mestre, o clock SCL é gerado pelo próprio PIC, no modo escravo, o clock e gerado por outro elemento atuando como mestre no barramento.

Após a escrita do dado, é retornado um valor *ack* indicando se a informação foi confirmada pelo destino: 1 indica que não foi confimada (nack) e 0 indica confirmação (ack).

Requisito: #use i2c

Exemplo:

```
i2c_start();
// inicia a comunicação
i2c_write(0xc1);
// endereça um dispositivo e ativa flag de leitura
conta = 0;
while (conta!=4)
{
   while (!i2c_poll()); // aguarda chegada dos dados
   buffer[conta++] = i2c_read();
   // armazena o byte no buffer
}
i2c_stop(); // finaliza comunicação
```

I2C_POLL()

Verifica se o módulo SSP ou MSSP recebeu um byte.

Sintaxe:

```
res = i2c_poll ()
```

Em que:

- *res* é um valor booleano

Esta função deve ser utilizada somente com o hardware interno SSP ou MSSP e retorna 1(true) no caso de haver um byte no buffer de recepção do módulo ou 0 (false) no caso negativo.

No caso de haver um caractere no buffer, uma chamada à função I2C_READ retorna imediatamente o caractere recebido.

Requisito: #use i2c

Exemplo:

```
i2c_start();
// inicia a comunicação
i2c_write(0xc1);
// endereça um dispositivo e ativa flag de leitura
conta = 0;
while (conta!=4)
{
   while (!i2c_poll()); // aguarda chegada dos dados
   buffer[conta++] = i2c_read();
   // armazena o byte no buffer
}
i2c_stop(); // finaliza comunicação
```

11.15. Comunicação SPI

As funções de comunicação serial síncrona utilizando o protocolo SPI somente estão disponíveis nos dispositivos dotados de módulo SSP ou MSSP.

SETUP_SPI()

Configura o módulo SSP ou MSSP para comunicação SPI.

Sintaxe:

```
setup_spi ( modo )
```

Em que:

- *modo* é um valor inteiro de 8 bits

Esta função inicializa o módulo SSP ou MSSP para o modo de comunicação SPI.

O argumento modo pode ser uma das seguintes constantes:

- SPI_L_TO_H: dados transmitidos na subida de SCK;
- SPI_H_TO_L: dados transmitidos na descida de SCK;
- SPI_MASTER: modo SPI mestre;

- SPI_SLAVE: modo SPI escravo, pino SS como entrada de seleção;
- SPI_SS_DISABLED: modo SPI escravo, pino SS como E/S;
- SPI_CLK_DIV_4: clock SPI = Fosc / 4;
- SPI_CLK_DIV_16: clock SPI = Fosc / 16;
- SPI_CLK_DIV_64: clock SPI = Fosc / 64;
- SPI_CLK_T2: clock SPI = TMR2 / 2.

As constantes dos mesmos grupos anteriores podem ser agrupadas pelo operador OR (|).

Exemplo:

```
setup_spi ( SPI_MASTER | SPI_L_TO_H | SPI_CLK_T2 );
```

SPI_READ()

Verifica se um dado foi recebido pela interface SSP ou MSSP no modo SPI.

Sintaxe:

```
res = spi_data_is_in ( dado )
```

Em que:

- *res* é um inteiro
- *dado* é um valor inteiro opcional

Esta função efetua a leitura de um dado por meio da interface SPI do PIC.

Se um dado for incluído, ele será primeiramente transferido para o dispositivo externo e depois será lido.

Observe que se o modo de operação for o mestre, o próprio PIC gera o sinal de clock SCK. Neste caso, para iniciar uma recepção, é necessário uma escrita no registrador SSPBUF, o que pode ser feito chamando a função com argumento zero spi_read(0).

No caso do modo escravo, a função somente vai retornar após a recepção do dado.

Exemplo:

```
dado = spi_read( 0 );
```

SPI_WRITE()

Escreve um dado na interface SPI (módulo SSP ou MSSP).

Sintaxe:

```
Spi_write ( dado )
```

Em que:

- *dado* é um valor inteiro

Exemplo:

```
dado = 10;
spi_write( dado );
```

SPI_DATA_IS_IN()

Verifica se um dado foi recebido pela interface SSP ou MSSP no modo SPI.

Sintaxe:

```
res = spi_data_is_in ()
```

Em que:

- *res* é um valor booleano

Esta função retorna 1 (true) se um dado tiver sido recebido pela interface SSP ou MSSP. Caso contrário, a função retorna 0 (false).

Exemplo:

```
while ( spi_data_is_in() && input (pin_b2));
if ( spi_data_is_in() ) dado = spi_read();
```

Anotações

Capítulo 12

Tópicos Avançados

12.1. Escrevendo Código Eficiente em C

Um dos principais aspectos da programação em alto nível é a otimização e eficiência do código gerado pelo compilador.

Quando programamos dispositivos tão limitados como os microcontroladores, a otimização pode ser fundamental para atender a requisitos de velocidade e tamanho de código.

Por isso estudaremos neste tópico algumas formas eficazes de escrever programas eficientes em C.

12.1.1. Utilização de Variáveis Booleanas

Muitas vezes utilizamos em um programa variáveis para o armazenamento de valores booleanos (Verdadeiro ou Falso, 0 ou 1).

Em sistemas mais poderosos, em que a memória RAM é normalmente medida em kilobytes ou Megabytes, não há qualquer problema em alocar uma posição inteira de memória para o simples armazenamento de uma condição booleana, mas no presente caso, isto não é verdade.

Não é necessário muito esforço para perceber que o uso de variáveis de bit ou booleanas pode trazer uma grande economia de memória RAM, além de permitir uma maior eficiência geral do programa.

Como exemplo dos ganhos que podem ser obtidos, vejamos dois casos de utilização de variáveis puramente booleanas: um com o uso de variáveis do tipo inteiro (8 bits) e outro com variáveis de bit:

Exemplo 12.1

```
#include <16f628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
int x,y,z;
main()
{
   x = 1;
   y = 0;
   if (x && !y) z = x;
}
```

O programa do exemplo 12.1, ao ser compilado, gera o código apresentado em seguida (somente a parte relevante é apresentada). Repare ainda que o tamanho total do programa é 25 words e a quantidade total de memória RAM utilizada pelo programa é 8 bytes:

Exemplo 12.2

```
.....................     x = 1;
0009:  MOVLW  01
000A:  MOVWF  x
.....................     y = 0;
000B:  CLRF   y
.....................     if (x && !y) z = x;
000C:  MOVF   x,F
000D:  BTFSC  STATUS.2
000E:  GOTO   014
000F:  MOVF   y,F
0010:  BTFSS  STATUS.2
0011:  GOTO   014
0012:  MOVF   x,W
0013:  MOVWF  z
```

Vejamos agora o mesmo programa escrito utilizando variáveis de bit (short in, int1 ou boolean):

Exemplo 12.3

```
#include <16f628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
short int x,y,z;
main()
{
   x = 1;
   y = 0;
   if (x && !y) z = x;
}
```

O programa do exemplo 12.3, ao ser compilado, gera o código assembly visto parcialmente em seguida. O tamanho total do programa é 23 words e o consumo de RAM caiu para 6 bytes. Repare que o compilador otimizou o código utilizando as instruções assembly de manipulação de bit, disponíveis no conjunto de instruções do PIC:

Exemplo 12.4

```
.....................     x = 1;
0009:  BSF    x
.....................     y = 0;
000A:  BCF    y
.....................     if (x && !y) z = x;
000B:  BTFSS  25.0
000C:  GOTO   012
000D:  BTFSC  25.1
000E:  GOTO   012
000F:  BCF    25.2
0010:  BTFSC  x
0011:  BSF    z
```

12.1.2. Testes Condicionais

Os testes condicionais também merecem uma atenção especial, já que o conjunto de instruções dos PICs beneficia o teste de zero de um registrador. É muito mais eficiente testar um registrador de forma booleana, do que efetuar a comparação relacional entre o registrador e o número zero.

No exemplo seguinte podemos observar as duas situações, em que um registrador é testado contra zero:

Exemplo 12.5

```
#include <16f628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP
int x,y;
main()
{
   x = 1;
   if (x == 1) y = 1;
}
```

Observe que o código da função **main()** anterior, após compilado, irá produzir o seguinte código assembly. Repare que são necessárias oito instruções para implementar as duas linhas de código C:

Exemplo 12.6

```
....................      x = 1;
0009:  BSF    x
....................      if (x == 1 ) y = 1;
000A:  MOVLW  00
000B:  BTFSC  x
000C:  MOVLW  01
000D:  SUBLW  01
000E:  BTFSS  STATUS.2
000F:  GOTO   011
0010:  BSF    y
```

Se utilizarmos um teste booleano para avaliar a variável, em vez do teste relacional do exemplo 12.5, podemos ter um grande ganho de eficiência no programa. Vejamos:

Exemplo 12.7

```
#include <16f628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOMCLR,NOLVP

int x,y;
main()
{
   x = 1;
   if (x) y = 1;
}
```

O código compilado da função **main()** anterior é apresentado em seguida. Repare que neste caso, foram necessárias apenas quatro instruções assembly para a implementação das duas linhas C anteriores, ou seja, um ganho de 50% sobre o exemplo utilizando a abordagem relacional.

Exemplo 12.8

```
....................      x = 1;
0009:  BSF    x
....................      if (x) y = 1;
000A:  BTFSS  25.0
000B:  GOTO   00D
000C:  BSF    y
```

12.1.3. Rotinas Matemáticas

Devido às limitadas capacidades matemáticas da ULA dos PICs, a realização de operações aritméticas diferentes da soma / subtração de 8 bits pode demandar um grande tempo de execução, além do espaço na memória de programa.

Por isso, devemos evitar a realização de operações matemáticas complexas e especialmente utilizando tipos de dados **float**.

Quando tais operações forem inevitáveis, aqui estão algumas sugestões para acelerar a realização de cálculos matemáticos com PICs:

1) Evite o uso de variáveis de 32 ou 16 bits;

2) Utilize, sempre que possível, divisões e multiplicações inteiras e se possível, utilizando múltiplos de potências de dois (já que essas operações podem ser facilmente traduzidas em rotações à direita ou esquerda, conforme o caso);

3) Evite o uso de valores fracionários; prefira o uso de valores inteiros. Assim, para valores entre 0,00 e 5,00, poderíamos utilizar uma variável de 16 bits com valores entre 0 e 500.

12.1.4. Utilizando Assembly no Código C

Por mais eficientes que sejam uma linguagem de programação e um compilador, dificilmente ele geram um código tão eficiente quanto um bom programador assembly.

Em aplicações críticas, nas quais cada microssegundo conta, podemos utilizar código assembly dentro do programa em C, de forma a acelerar e/ou otimizar a sua execução.

De maneira geral, podemos dizer que os principais alvos deste tipo de otimização são: rotinas de tratamento de interrupção (nos casos da ocorrência de um elevado número de interrupções num curto espaço de tempo), rotinas de manipulação de dados como conversões binário/decimal/hexadecimal, dentre outras.

A linguagem assembly pode ser inserida no código C em qualquer ponto, utilizando as diretivas **#asm** e **#endasm**. As variáveis C conservam seus nomes na utilização da linguagem assembly.

No próximo tópico veremos algumas formas de otimização de tratamento de interrupções utilizando código assembly.

12.2. Entrada e Saída em C

O pressuposto principal de um sistema ou aplicação envolvendo microcontroladores é a interação com o ambiente externo ao chip, sendo assim, o estudo das técnicas de interface de entrada e saída é de suma importância na programação de tais dispositivos em linguagem C.

Os compiladores CCS incluem um suporte muito eficiente à manipulação das portas de entrada e saída dos PICs. As funções utilizadas para acesso às portas dos PICs estão relacionadas em seguida e já foram vistas no capítulo 11:

```
output_low()
output_high()
output_float()
output_bit()
input()
output_X()
input_X()
port_b_pullups()
set_tris_X()
```

Neste tópico, vamos observar o funcionamento das funções anteriores em relação aos três modos de funcionamento do tratamento de E/S existentes no compilador e especificados pelas seguintes diretivas:

```
#use fast_io (porta)
#use fixed_io (porta_outputs = pinos)
#use standard_io (porta)
```

12.2.1. Modo Padrão

Como já foi visto, o modo padrão de manipulação de E/S é o "standard_io" no qual o compilador adiciona código para configurar o registrador TRIS, de forma a tornar o pino uma entrada ou saída de acordo com a instrução executada.

Assim, se executarmos a seguinte linha de código:

```
output_high (pin_b0);
```

O compilador gera código para primeiramente configurar o pino RB0 como uma saída e em seguida, seta o bit 0 do registrador PORTB, conforme o código em seguida:

```
BSF     STATUS.5
BCF     TRISB.0
BCF     STATUS.5
BSF     PORTB.0
```

Note que neste modo, o compilador sempre gera código de configuração do registrador TRIS, mesmo que o acesso seja feito seqüencialmente.

A desvantagem óbvia deste modo de manipulação de E/S é que em aplicações com uso intensivo das portas de E/S, o código gerado pode ser muito grande e potencialmente mais lento que nos modos fixo ou rápido.

Como vantagem deste modo, podemos destacar a facilidade de uso, já que o compilador encarrega-se de configurar o pino de acordo com a operação realizada.

Uma outra característica interessante deste modo de funcionamento é que o pino permanece sempre no seu último estado configurado. Por isso podemos utilizar um comando do tipo:

```
input (pin_b0);
```

Para configurar o pino RB0 como entrada, na certeza de que ele permanecerá assim configurado indefinidamente ou até que se faça uma escrita no pino.

12.2.2. Modo Fixo

O acesso à E/S no modo fixo funciona de forma diferente do modo padrão.

Neste modo, utiliza-se a diretiva:

```
#use fixed_io (porta_outputs = pinos)
```

Para determinar ao compilador que os pinos especificados serão utilizados somente como saída. Neste caso, a configuração da direção dos pinos da porta especificada será fixa, ou seja, os pinos definidos como saídas funcionam apenas como saídas e os definidos como entradas, apenas como entradas.

Isto significa que o respectivo registrador TRIS será programado conforme especificado na diretiva, não sendo essa configuração alterada de acordo com a natureza da operação desempenhada pela função utilizada pelo programador.

Vejamos um exemplo prático. Suponha o seguinte fragmento de programa:

```
boolean x;
#use fixed_io(c_outputs = pin_c0, pin_c1)
output_low(pin_c0);
x = input(pin_c0);
output_low(pin_c2);
```

Veja que o primeiro comando de saída "output_low(pin_c0)" realmente coloca o pino RC0 em nível lógico 0, mas o próximo comando "x=input(pin_c0)" não irá configurar o pino como entrada. Ao contrário disso, o pino será novamente configurado como saída, resultando em que "x" será a leitura do estado da saída do pino RC0.

O último comando "output_low(pin_c2)" também não produz o efeito desejado, já que o pino RC2 foi configurado como entrada pela diretiva #use fixed_io. Neste caso, será escrito o valor zero no bit 2 do registrador PORTC, no entanto esse pino continuará configurado como entrada.

Observe que o compilador sempre adiciona código de configuração de direção do pino, tal como no modo padrão.

12.2.3. Modo Rápido

Outro modo de manipulação de E/S disponível nos compiladores CCS é o modo rápido (fast I/O), no qual o compilador não gera código de configuração da direção para os registradores TRIS. Isto torna o acesso às portas de E/S substancialmente mais rápido, no entanto o programador deve providenciar todo o código necessário para configurar os registradores TRIS envolvidos.

12.2.4. Outra Modalidade de Acesso

Além dos modos descritos anteriormente, existem ainda outras formas de acesso ao hardware, mas sem a intervenção do compilador.

Uma forma muito utilizada consiste na criação de variáveis para acesso direto aos registradores TRISx e PORTx (diretiva #byte), ou para acesso direto a bits específicos (utilizando a diretiva #bit).

Esta forma de acesso assemelha-se muito à utilizada em assembly, com a vantagem de que não é necessário se preocupar com a troca de bancos de memória, pois o compilador realiza tais operações automaticamente.

Veja em seguida um exemplo de utilização desta modalidade de acesso para inverter o bit 0 da porta B em um PIC da série 16:

Exemplo 12.9

```
#byte portb = 0x06
#bit rb0 = portb.0

rb0 = !rb0;  // inverte o estado do pino RB0
...
```

12.3. Interrupções em C

O compilador CCS prevê duas formas básicas para o tratamento de eventos de interrupção:

- uma modalidade que poderíamos chamar de "automática" na qual o compilador gera praticamente todo o código necessário ao tratamento de interrupções (verificação de evento, salvamento e restauração de registradores, apagamento de flags de interrupção, etc.), cabendo ao programador somente a inclusão de funções de tratamento dos eventos individuais;
- e outra modalidade "manual", na qual o programador deve incluir todo o código para tratamento de interrupções.

A diferença entre as duas modalidades reside principalmente na otimização e facilidade de uso: a primeira é de uso mais simples, mas também gera um código muito maior, o que em aplicações críticas em tempo pode inviabilizar o programa. Já a segunda produz um código normalmente menor, mas também outorga ao programador a tarefa de realizar os diversos procedimentos necessários ao tratamento de interrupções.

Lembre-se que em ambos os casos, o bit GIE é automaticamente desligado pelo processador ao desviar para o vetor de interrupção e ligado novamente pela execução da instrução RETFIE (que o compilador insere automaticamente ao final de uma função definida para o tratamento de interrupção).

Vejamos então o funcionamento de cada modalidade de tratamento de interrupções.

12.3.1. Tratamento "Automático"

Nesta modalidade, como já dissemos, o compilador gera todo o código da rotina de tratamento de interrupção (RTI). A única tarefa que cabe ao programador é a construção das funções de tratamento individual dos eventos de interrupção.

Para utilizar esta modalidade de RTI, utilizamos as diretivas **#INT_xxx** antes de cada função destinada ao tratamento individual de interrupção.

Vejamos um exemplo de RTI para tratamento de eventos do timer 0.

Exemplo 12.10

```
...
int x;
...
#int_timer0
void trata_timer0(void)
{
   x++;    // incrementa a variável global "x"
}
```

Observe que não é necessário adicionar quaisquer códigos relativos ao apagamento do flag de interrupção (no caso o T0IF). O compilador faz isso automaticamente.

Esta modalidade de utilização facilita muito a criação de RTIs, mas também aumenta bastante o tamanho do programa gerado, além de aumentar a latência do tratamento efetivo da interrupção, ou seja, o tempo entre a ocorrência do evento de interrupção e a chamada da função definida pelo programador.

Por isso, em aplicações críticas de tempo, é preferível utilizar a modalidade "manual", que veremos em seguida.

12.3.2. Tratamento "Manual"

Nesta modalidade de tratamento de interrupção, utiliza-se apenas um tipo especial de diretiva do compilador: **#INT_GLOBAL**.

Esta diretiva deve ser colocada antes da função encarregada do tratamento de interrupção. Assim, o compilador automaticamente monta a função a partir do endereço do vetor de interrupção (0x0004 nos PICs da série 16, 0x0008 (alta prioridade) e 0x0018 (baixa prioridade) nos PICs da série 18).

Observe que nesse caso, o programador deve providenciar todo o código de salvamento e restauração de contexto (ou seja, salvar ao menos os registradores W, STATUS e outros que possam ser eventualmente modificados dentro da RTI).

Além disso, é necessário escrever código para verificar os flags de interrupção, a chamada da função específica de tratamento do evento de interrupção, além de apagar o flag responsável pela interrupção (no registrador INTCON ou PIRx).

Exemplo 12.11

```
// exemplo de utilização de tratamento "manual" de interrupções
int x;
#bit t0if = 0x0b.2

int w_temp, status_temp; // registradores temporários
#byte status = 0x03

#inline
void trata_t0()
// esta é a rotina de tratamento do evento de interrupção do timer 0
// a diretiva #inline determina que o código da função seja inserido
// no local de chamada da função
{
   x++;
   t0if=0;
}
```

```
void salva_contexto()
// salva registradores que podem ser alterados durante a RTI
{
   #asm
          movwf   w_temp
          swapf   status,w
          movwf   status_temp
   #endasm
}
void restaura_contexto()
// restaura o conteúdo dos registradores para o seu estado
// antes da RTI
{
   #asm
          swapf   status_temp,w
          movwf   status
          swapf   w_temp,f
          swapf   w_temp,w
   #endasm
}


#int_global
void trata_int()
// Rotina principal de tratamento de interrupção. Observe o uso da dire-
// tiva #int_global. Neste caso, não podemos utilizar as de tratamento
// individual de interrupção
{
   salva_contexto();
   if (t0if) trata_t0();
   restaura_contexto();
}
```

12.3.3. Priorização de Interrupções

Além dos aspectos vistos até agora, um outro tópico importante a ser estudado é o da priorização de interrupções.

Como o leitor deve saber, em aplicações utilizando múltiplas interrupções, pode haver ocasiões em que ocorram dois os mais eventos de interrupção simultaneamente.

A ordem de tratamento das interrupções é determinada basicamente pela seqüência de verificação dos FLAGs de interrupção, ou seja, o primeiro evento (mais prioritário) é o colocado no topo da rotina de tratamento de interrupções.

Mas como podemos estabelecer a estrutura hierárquica de avaliação de interrupções na linguagem C?

A resposta existe e varia conforme a modalidade de tratamento de interrupção utilizada.

Na modalidade "manual", a ordem de prioridade de tratamento de interrupções pode ser estabelecida da mesma forma como em assembly: a primeira interrupção verificada será a de maior prioridade.

Na modalidade "automática", podemos utilizar a diretiva **#priority** para definir para o compilador a ordem de avaliação das interrupções.

Assim, supondo a utilização das interrupções do timer 0 e timer 1, por exemplo, podemos utilizar a seguinte diretiva para definir a ordem de prioridade de tratamento das interrupções:

```
#priority timer1, timer0
```

Neste caso, a interrupção do timer 1 seria prioritária em relação à do timer 0.

Note que nos PICs da série 18, onde encontramos hardware interno que permite a configuração entre dois níveis de prioridade distintos para cada fonte de interrupção interna (com exceção da INT0), podemos utilizar a opção FAST junto às diretivas #INT_XXX, para determinar ao compilador que configure quele tipo de interrupção para alta prioridade.

12.4. Utilizando os Timers Internos

Neste tópico vamos verificar como configurar os timers internos para a geração de interrupções em intervalos fixos de tempo.

12.4.1. Timer 0

Vejamos como configurar o módulo timer 0 disponível em todos os PICs da série Midrange (série 16) para a geração de interrupções a cada 1 segundo.

Considerando o clock do processador como sendo de 4MHz e utilizando o prescaler, dividindo por 64, teremos uma freqüência de entrada no timer 0 de 15625 Hz.

Se programarmos o timer para dividir esse sinal por 125, teremos um sinal de saída de exatamente 125 Hz.

Para programar o timer 0 para dividir o sinal por 125, basta carregá-lo a cada estouro de contagem com o valor 131 (256 - 125).

Vejamos um programa para piscar um led no pino 0 da porta B a uma freqüência de 1Hz, utilizando a interrupção do timer 0:

Exemplo 12.12

```
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOLVP


#int_timer0
void trata_t0 ()
{
   static boolean led;
   static int conta;
   // reinicia o timer 0 em 131 menos a contagem que já passou
   set_timer0(131-get_timer0());
   conta++;
   // se já ocorreram 125 interrupções
   if (conta == 125)
   {
         conta=0;
         led = !led;  // inverte o led
         output_bit (pin_b0,led);
   }
}

main()
{
   // configura o timer 0 para clock interno e prescaler dividindo por 64
   setup_timer_0 ( RTCC_INTERNAL | RTCC_DIV_64 );
   set_timer0(131); // inicia o timer 0 em 131
   // habilita interrupções
   enable_interrupts (global | int_timer0);
   while (true); // espera interrupção
}
```

12.4.2. Timer 1

O módulo timer 1, disponível em alguns modelos da série 16 e em todos da série 18, é um contador / temporizador de 16 bits, com capacidade de ser ligado/desligado por software e prescaler independente.

Vejamos um exemplo de configuração do timer 1 em um programa para piscar um led conectado ao pino RB0 de um PIC 16F877.

A menor freqüência de interrupção do TMR1 que podemos obter com um clock de 4MHz é 4MHz / 4 / 8 / 65536 = 1,9073 Hz.

Para o nosso exemplo utilizaremos uma freqüência de 2Hz, o que implica em que o timer 1 deve dividir a freqüência por 62500. Assim, devemos inicializá-lo com o valor 3036 (65536 - 62500).

Vejamos um programa para realizar esta tarefa:

Exemplo 12.13

```
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOLVP

#int_timer1
void trata_t1 ()
{
   static boolean led;
   static int conta;
   // reinicia o timer 1 em 3036 menos a contagem que já passou
   set_timer1(3036-get_timer1());
   conta++;
   // se já ocorreram 2 interrupções
   if (conta == 2)
   {
         conta=0;
         led = !led;  // inverte o led
         output_bit (pin_c2,led);
   }
}

main()
{
   // configura o timer 1 para clock interno e prescaler dividindo por 8
   setup_timer_1 ( T1_INTERNAL | T1_DIV_BY_8 );
   // inicia o timer 1 em 3036
   set_timer1(3036);
   // habilita interrupções
   enable_interrupts (global );
   enable_interrupts (int_timer1);
   while (true); // espera interrupção
}
```

12.4.3. Obtendo Mais Precisão dos Timers

Uma técnica muito interessante para implementar bases de tempo precisas com os timers internos dos PICs é a chamada acúmulo de erros.

Esta técnica consiste em implementar um contador de software que é decrementado a cada ciclo de interrupções, e uma vez que o seu valor atinja um determinado limite, subtrai-se a freqüência de entrada, mantendo-se o resto no registrador.

Vejamos um exemplo prático: suponha que seja necessário implementar uma base de tempo de 1 segundo utilizando o timer 0 num PIC com Fosc = 4MHz.

O primeiro passo é selecionar o maior fator de divisão do prescaler que resulte em uma freqüência inteira. No nosso caso, o maior fator de divisão é 64, resultando em uma freqüência de 15625 Hz.

Este valor (15625) será carregado em uma variável de contagem e a cada ciclo de interrupção, subtrai-se 256 dessa variável. O valor 256 é utilizado pois consiste no fator de divisão natural do timer 0, que nos PICs da série 16 é sempre de 8 bits.

Se o valor dela for igual ou menor que zero, soma-se 15625 à variável, executam-se os comandos relativos ao evento de passagem de 1 segundo e o processo reinicia.

Observe que o resto presente na variável de contagem (erro residual da divisão) é acumulado durante a contagem, reduzindo o erro total a praticamente zero.

O inconveniente deste técnica é que o intervalo de tempo entre cada evento de tempo (no caso, a passagem de 1 segundo) não é constante, o que em alguns casos pode ser inaceitável.

Vejamos um exemplo para piscar um LED conectado ao pino RB0 a uma freqüência de 1 Hz.

Exemplo 12.14

```
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOLVP

signed long int t0_conta;

#int_timer0
void trata_t0 ()
{
   static boolean led;
   t0_conta -= 256; // subtrai 256 da contagem
   if (t0_conta<=0) // se a contagem é igual ou menor que zero
   {
         // soma 15625 ao valor da contagem
         t0_conta += 15625;
         led = !led;  // inverte o led
         output_bit (pin_c2,led);
   }
}

main()
{
   t0_conta = 15625;
   // configura o timer 0 para clock interno e
   // prescaler dividindo por 64
   setup_timer_0 ( RTCC_INTERNAL | RTCC_DIV_64 );
   // habilita interrupções
   enable_interrupts (global | int_timer0);
   while (true); // espera interrupção
}
```

12.5. Comunicando-se com o PIC

Em muitas aplicações microcontroladas, pode ser necessário realizar a comunicação entre o microcontrolador e um ou mais dispositivos externos.

Esses dispositivos podem estar localizados tanto na mesma placa do circuito, como fora dela, a metros ou mesmo dezenas de quilômetros de distância.

A escolha do sistema de comunicação mais adequado para realizar a tarefa depende de diversos fatores, como: velocidade, imunidade a ruídos, custo, etc.

As técnicas de comunicação podem ser divididas em duas grandes categorias: serial e paralela.

Na comunicação serial, a informação a ser transmitida é fracionada em pequenas partes (bits) que são enviadas ao equipamento receptor uma após a outra, em série, daí a denominação de comunicação serial. Como exemplos de sistemas de comunicação serial podemos citar: as interfaces seriais dos computadores (RS-232, USB, FIREWIRE), protocolos de redes locais (ETHERNET, TOKEN-RING, etc.), outros protocolos como I^2C, SPI, 1-WIRE, LIN, CAN, etc.

Na comunicação paralela, os bits componentes da informação são transmitidos simultaneamente (total ou parcialmente) em paralelo. Como exemplos de sistemas de comunicação paralela, podemos citar os barramentos internos dos microprocessadores e microcontroladores, barramento ISA, PCI, VESA, AGP, a interface de impressora paralela dos microcomputadores (também chamada de Centronics), SCSI, IDE, etc.

A escolha dentre os tipos de comunicação é um paradoxo da dualidade custo x benefício: na comunicação paralela, temos uma alta velocidade de comunicação, mas também uma alta utilização de meios de transmissão, além de uma baixa imunidade a ruídos. Na comunicação serial, temos uma velocidade máxima menor que na comunicação paralela, mas também uma menor utilização de meios de transmissão e uma melhor imunidade a ruídos.

Na verdade, as técnicas de comunicação serial vêm sendo continuamente aprimoradas, e atualmente temos diversos protocolos de comunicação capazes de atingir altíssimas velocidades, tais como: o USB 2.0, Firewire, SCSI serial (Serial Attached SCSI - SAS, Fibre Channel Protocol - FCP, Serial Storage Architecture - SSA), o novo Serial ATA, etc.

Os protocolos seriais podem ser classificados ainda em duas categorias: síncronos e assíncronos.

Nos protocolos síncronos, além da(s) linha(s) de comunicação, encontramos uma ou mais linhas de sincronização (clock). A informação na linha de dados é transmitida a cada transição da linha de clock. Neste tipo de protocolo, encontramos ainda uma outra classificação para os elementos de

comunicação: o elemento que gera o sinal de sincronização de transmissão (clock) é chamado de mestre, o(s) dispositivo(s) que recebe(m) o sinal de sincronismo é(são) chamado(s) de escravo(s).

Já nos protocolos assíncronos, não encontramos uma linha específica para sincronização. Neste caso, a sincronização entre os elementos transmissor e receptor é garantida pela precisão dos clocks de cada elemento e também pela utilização de sinais marcadores de início (start) e fim (stop) da palavra transmitida.

12.5.1. Comunicação Serial Assíncrona

Como já dissemos, a comunicação assíncrona caracteriza-se pela utilização de marcadores de início e fim de transmissão.

Na figura 12.1, podemos observar a transmissão de um sinal serial assíncrono (no caso o caractere 'A'):

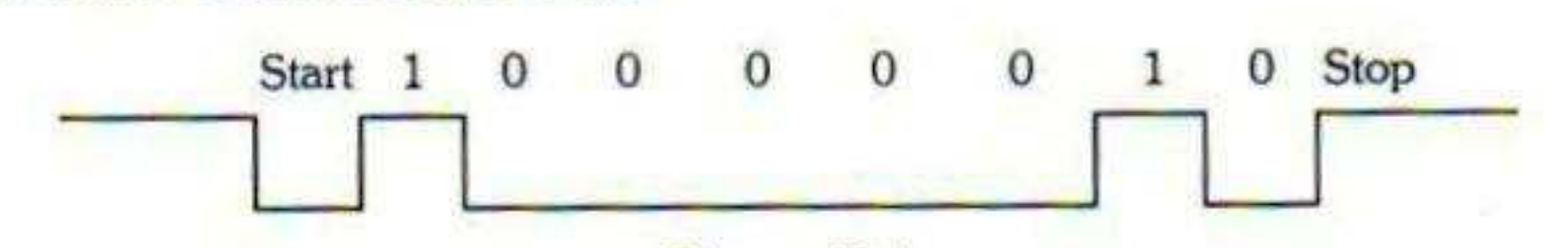

Figura 12.1

Observe que a transmissão começa por um sinal de início (start), seguido pelos bits de dados (no caso 8 bits), iniciando pelo bit LSB e após eles, um bit de parada (stop) para sinalizar o fim do caractere.

Existem diversas formas de implementação de comunicação serial assíncrona, tanto por hardware quanto por software, e apesar de termos um excelente suporte interno à comunicação assíncrona nos compiladores CCS, ainda assim apresentamos um conjunto de funções de comunicação que pode ser utilizado tanto nos compiladores CCS quanto em outros compiladores (mediante pequenas modificações).

12.5.1.1. Comunicação Assíncrona por Software

Em seguida, apresentamos uma rotina de transmissão e recepção serial assíncrona, toda baseada em software.

Neste tipo de rotina, também denominada *"bit bang"*, a temporização necessária à geração do sinal serial é realizada por rotinas de atraso dentro do próprio programa.

Isto significa que durante a transmissão ou recepção de caracteres não é possível realizar outras tarefas no software, o que limita bastante a sua aplicação, no entanto esta é uma boa opção para incluir uma simples rotina de comunicação em chips que não tenham USART interna.

Uma forma mais eficiente de implementação é utilizar um dos timers internos dos PICs para a tarefa de temporização. Neste caso, a rotina de transmissão / recepção seria embutida na rotina de tratamento de interrupção do respectivo timer.

Vejamos uma implementação sem o uso de interrupções:

```
/*****************************************************************/
/*  RS232.C     Biblioteca de Comunicação serial assíncrona por  */
/*                              software                         */
/*                                                               */
/*  Autor: Fabio Pereira                                         */
/*                                                               */
/*  Velocidade : 9600 Bps, 1 Start, 1 Stop, Sem paridade         */
/*                                                               */
/*  Estas rotinas funcionam sem modificação para velocidades de 9600 e */
/*  19200 Bps.                                                   */
/*                                                               */
/*****************************************************************/

// Definições de comunicação
/*
   Para alterar a velocidade de comunicação, basta alterar o valor da
   constante baud_rate
*/
#ifndef baud_rate
  const long int baud_rate = 9600;
#endif
const int tempo_bit_dado = 1000000/baud_rate-10; // tempo do bit de dado
const int tempo_bit_start = 1500000/baud_rate;  // tempo do bit de start

// Definições dos pinos de comunicação
// Para utilizar outros pinos, basta incluir novas definições
// no arquivo do programa onde esta biblioteca for incluída
#bit pino_tx = 0x06.0    // pino de transmissão é o RB0
#bit pino_rx = 0x06.1    // pino de recepção é o RB1
#bit dir_tx  = 0x86.0    // direção do pino de tx
#bit dir_rx  = 0x86.1    // direção do pino de rx

void rs232_inicializa (void)
{
   dir_tx = 0;     // pino de tx como saída
   pino_tx = 1;    // coloca o pino de tx em nível 1
   dir_rx = 1;     // configura o pino de rx como entrada
}
void rs232_transmite (char dado)
{
   boolean result;
   int conta;
   // primeiro o bit de start
   pino_tx = 0;
   delay_us(tempo_bit_dado);      // aguarda o tempo de start
   conta = 8;                     // são 8 bits de dados
   while (conta)
   {
         // desloca o dado à direita e dependendo do resultado
         // seta ou não a saída
         if (shift_right ( &dado, 1, 0)) pino_tx=1; else pino_tx=0;
         delay_us ( tempo_bit_dado);   // aguarda o tempo do bit de dado
```

```
        conta--;        // decrementa um no número de bits a transmitir
    }
    pino_tx = 1;                    // seta a saída TX (bit de stop)
    delay_us(tempo_bit_dado);       // aguarda o tempo de 1 bit
}

char rs232_recebe (void)
{
    int conta,dado;
    while (pino_rx);                // aguarda o bit de start
    delay_us(tempo_bit_start);      // aguarda o tempo de start
    conta = 8;                      // são 8 bits de dados
    dado = 0;                       // apaga o dado recebido
    while (conta)
    {
        shift_right( &dado, 1, pino_rx);     // insere o bit recebido
                                             // deslocando à direita a
                                             // variável com o dado
                                             // recebido
        delay_us(tempo_bit_dado);            // aguarda o tempo de 1 bit
        conta--;                        // decrementa o número de bits
    }
    delay_us(tempo_bit_dado);           // aguarda o tempo de 1 bit
    return dado;                        // retorna o dado recebido
}
```

Exemplo 12.15

```
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP
#include <rs232.c>
main()
{
    // programa de teste: ecoa o dado recebido na porta serial
    // TX = RB0 e RX = RB1
    int x = tempo_bit_dado;
    rs232_inicializa ();
    while (true)
    {
        rs232_transmite(rs232_recebe());
    }
}
```

12.5.1.2. Comunicação Assíncrona por Hardware

Nos dispositivos que contam com USART interna, podemos utilizar um simples conjunto de funções C para o acesso a ela, conforme veremos em seguida.

Novamente, a rotina apresentada não utiliza a capacidade de interrupção da USART. Uma alternativa é implementar rotinas de tratamento de interrupção para os eventos TXIF e RCIF, mas este tipo de implementação varia conforme a aplicação utilizada, fugindo assim ao escopo genérico deste livro.

```
//*****************************************************************
//*                                                               *
//*  USART.C                                                      *
//*                                                               *
//*  Biblioteca de manipulação da USART                           *
//*  Suporta o modo assíncrono                                    *
//*                                                               *
//*  Autor: Fábio Pereira                                         *
//*                                                               *
//*****************************************************************

char usart_rx, usart_tx, txreg, rcreg, spbrg;

struct rcsta_reg
   {
          int rx9d : 1;
          int oerr : 1;
          int ferr : 1;
          int aden : 1;
          int cren : 1;
          int sren : 1;
          int rx9  : 1;
          int spen : 1;
   } rcsta;

struct txsta_reg
   {
          int tx9d : 1;
          int trmt : 1;
          int brgh : 1;
          int xxx  : 1;
          int sync : 1;
          int txen : 1;
          int tx9  : 1;
          int csrc : 1;
   } txsta;

// define os endereços das variáveis
#locate rcsta = 0x18
#locate txreg = 0x19
#locate rxreg = 0x1a
#locate txsta = 0x98
#locate spbrg = 0x99
#byte r_pir1  = 0x0c      // define o registrador r_pir1
#bit flag_rc = r_pir1.5  // define o flag_rc
// definições utilizadas nas funções

void usart_inicializa ( int vel, boolean brgh )
/*
   O valor dos parâmetros vel e brgh deve ser retirado a partir das
tabelas de baud rate fornecidas pela Microchip ou no livro:
Microcontroladores PIC: Técnicas Avançadas.
*/

{
   txsta.brgh = brgh;      // seleciona o modo do gerador de baud rate
   spbrg = vel;            // configura o gerador de baud rate

   // configura os pinos da USART como entradas !!!!!

   #if __device__ == 627
```

```
input (pin_b1);
input (pin_b2);
#endif;
#if __device__ == 628
input (pin_b1);
input (pin_b2);
#endif;
#if __device__ == 876
input (pin_c7);
input (pin_c6);
#endif
#if __device__ == 877
input (pin_c7);
input (pin_c6);
#endif

   txsta.sync = 0;                // seleciona o modo assíncrono
   rcsta.spen = 1;                // habilita a USART
   txsta.tx9 = 0;                 // seleciona o modo de 8 bits
   txsta.txen = 1;                // ativa o transmissor da USART
   rcsta.cren = 1;                // modo de recepção contínua
}

void usart_transmite (char dado)
{
   while (!txsta.trmt);   // aguarda o buffer de transmissão esvaziar
   txreg = dado;          // coloca novo caractere para transmissão
}

char usart_recebe (void)
{
   while (!flag_rc);      // aguarda a recepção de caracteres
   return rxreg;          // retorna o caractere recebido
}
```

Exemplo 12.16

```
// Este programa irá ecoar na tela do terminal os caracteres digitados
// pelo usuário. Lembre-se que a transmissão / recepção é feita pelos
// pinos da USART !!!

#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP
#include <usart.c>
main()
{
   usart_inicializa (12,1); // velocidade: 19200
   usart_transmite ("testando USART \r\n");
   while (true) usart_transmite(usart_recebe());
}
```

12.5.2. Protocolo SPI

O SPI (Serial Peripheral Interface - Interface Serial de Periféricos) é um protocolo de comunicação síncrono de alta velocidade (até 3 Megabits por segundo) muito utilizado em conversores A/D, potenciômetros digitais, memórias seriais, etc.

Atualmente, existem inclusive memórias FLASH SPI com capacidade de até 256 Megabits, além de cartões de memória (MultimediaCard) com capacidade de até 64 Megabytes e compatíveis com SPI.

Uma variação do protocolo SPI é o Microwire, desenvolvido pela National Semiconductors. Esse protocolo é muito similar ao SPI com diferenças apenas na quantidade de bits transmitidos e na configuração da mensagem de controle.

O SPI foi desenvolvido originalmente pela Motorola e adotado por diversos fabricantes ao longo do tempo e conta com uma interface física constituída pelas seguintes linhas:

- CS ou SS: Chip Select (Seleção de Dispositivo): é utilizada para selecionar ou habilitar o dispositivo com o qual se deseja comunicar e também para encerrar a execução dos comandos transmitidos pelo dispositivo mestre;
- Clock ou SCLK: utilizada para sincronização entre o dispositivo mestre (aquele que gera o sinal de clock) e o dispositivo escravo (o que recebe o clock). O protocolo especifica que esse sinal deve ser simétrico (tempo alto igual ao tempo baixo), devendo a informação de saída estar disponível no mínimo 30 ns antes da borda de subida do clock e lida até 30ns antes da borda de descida do clock;
- SI ou MOSI: Serial In (Entrada Serial): linha de recepção de dados;
- SO ou MISO: Serial Out (Saída Serial): saída de dados do dispositivo.

Muitos projetistas optam por conectar as linhas DI e DO juntas, desta forma a quantidade de linhas de conexão entre o dispositivo mestre e o escravo é reduzida para um mínimo de três (CS, Clock e Dados).

A implementação do protocolo é muito simples, já que não existem facilidades de software para convivência de múltiplos dispositivos simultaneamente no mesmo barramento. A única previsão para coexistência de múltiplos dispositivos é física, na forma da entrada de seleção de chip (CS).

Outro detalhe importante sobre esse protocolo é a existência de quatro modos de operação. Cada modo caracteriza-se por apresentar polaridade (definida pela Motorola como CPOL) e fase (chamada CPHA) de clock distintas, conforme a tabela 12.1.

Observe ainda que nos modos 0 e 1, o clock é ativo alto e nos modos 2 e 3, o clock é ativo baixo.

Modo	CPOL	CPHA	Descrição
0	0	0	O dado é armazenado na borda de subida do clock.
1	0	1	O dado é armazenado na borda de descida do clock.
2	1	0	O dado é armazenado na borda de descida do clock.
3	1	1	O dado é armazenado na borda de subida do clock.

Tabela 12.1

Observe que é importante também verificar as especificações de operação de cada dispositivo, já que o protocolo não possui um padrão rigidamente definido.

Outra característica do protocolo SPI é que todas as operações são sempre precedidas por comandos de 8 bits, utilizados para descrever a natureza da operação.

Na tabela seguinte, podemos observar alguns dos comandos válidos para as memórias EEPROM da série 25xxxxx, bem como as suas funções:

Comando	Binário	Descrição
READ	0000 0011	Lê o dado da posição atual de memória
WRITE	0000 0010	Escreve um dado na posição atual da memória
WREN	0000 0110	Habilita escrita na memória
WRDI	0000 0100	Desabilita escrita na memória
RDSR	0000 0101	Lê o registrador de estado da memória
WRSR	0000 0001	Escreve no registrador de estado da memória

Tabela 12.2

Para encerrar ou completar a execução de um comando, o dispositivo mestre deve colocar a linha CS em nível "1", retornando-a ao nível "0" para o envio de novos comandos.

Na figura 12.2 temos a pinagem de uma típica memória SPI.

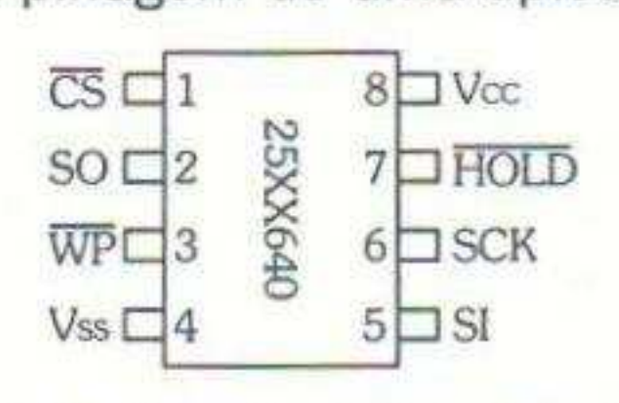

Figura 12.2 - Memória 25LC640 de 64Kbits.

12.5.2.1. Operação de Leitura

Uma operação de leitura é precedida pelo comando de leitura (READ - 0000 0011), seguindo pelos 16 bits do endereço da memória (iniciando pelo MSB).

Após o último bit do endereço, a memória passa a transmitir serialmente o dado relativo ao endereço especificado pela sua linha SO, iniciando pelo MSB.

Caso seja desejada a leitura de somente uma posição de memória, o dispositivo mestre deve, após a recepção do último bit do dado (o LSB), setar a linha CS, de forma a permitir que a memória possa aceitar novos comandos.

É possível ainda continuar a leitura de dados, já que as memórias SPI possuem um registrador de endereços interno, que é incrementado automaticamente após cada byte enviado para o dispositivo mestre.

Para continuar a leitura, basta continuar o envio de pulsos de clock (oito para cada novo byte a ser lido). Para encerrar a operação de leitura, basta novamente setar a linha CS após o último bit de dado ter sido recebido.

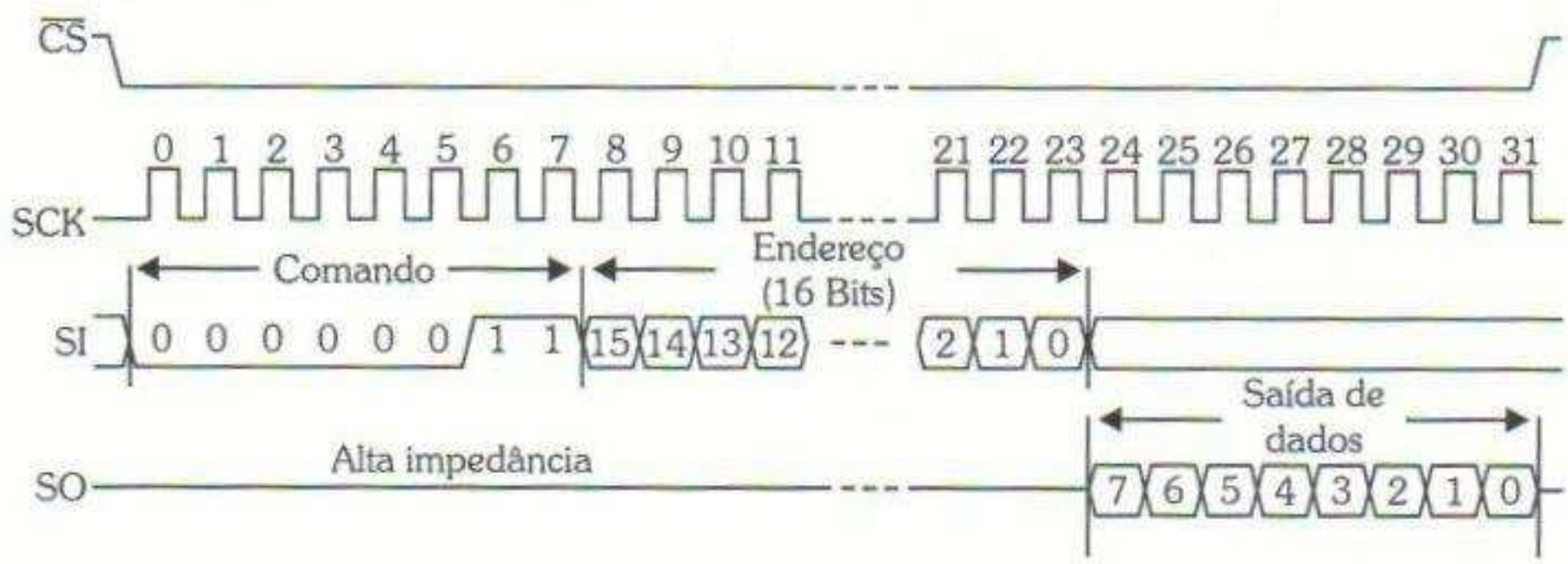

Figura 12.3

12.5.2.2. Operação de Escrita

A operação de escrita em memórias SPI é precedida pelo comando de escrita (WRITE - 0000 0010), seguida pelos 16 bits do endereço da memória em que será escrita a informação.

Após o último bit do endereço (LSB), o dispositivo mestre envia os 8 bits do dado a ser escrito, iniciando pelo MSB.

Repare que essas memórias aceitam um modo de escrita chamado de Page-Write (Escrita de página ou bloco) que permite que até 32 bytes sejam gravados simultaneamente, reduzindo significativamente o tempo necessário para gravação da memória.

Nesta modalidade, após a transmissão do primeiro byte, transmitem-se os bytes seguintes, um após o outro, até um máximo de 32 (desde que todos estejam localizados no mesmo bloco ou página de memória).

Em ambos os casos (escrita de byte simples ou blocos), a operação de gravação é levada a cabo ao elevar a linha CS ao nível "1".

Observe que este procedimento somente deve ocorrer após o último bit (o menos significativo) do dado ter sido transmitido para a memória.

Após o início da gravação, pode-se retornar a linha CS ao nível "0". O andamento da operação de escrita pode ser monitorado pelo bit 0 (WIP) do registrador de estado da memória: quando em "1", uma operação de escrita está em andamento, em "0", não há operação de escrita em andamento.

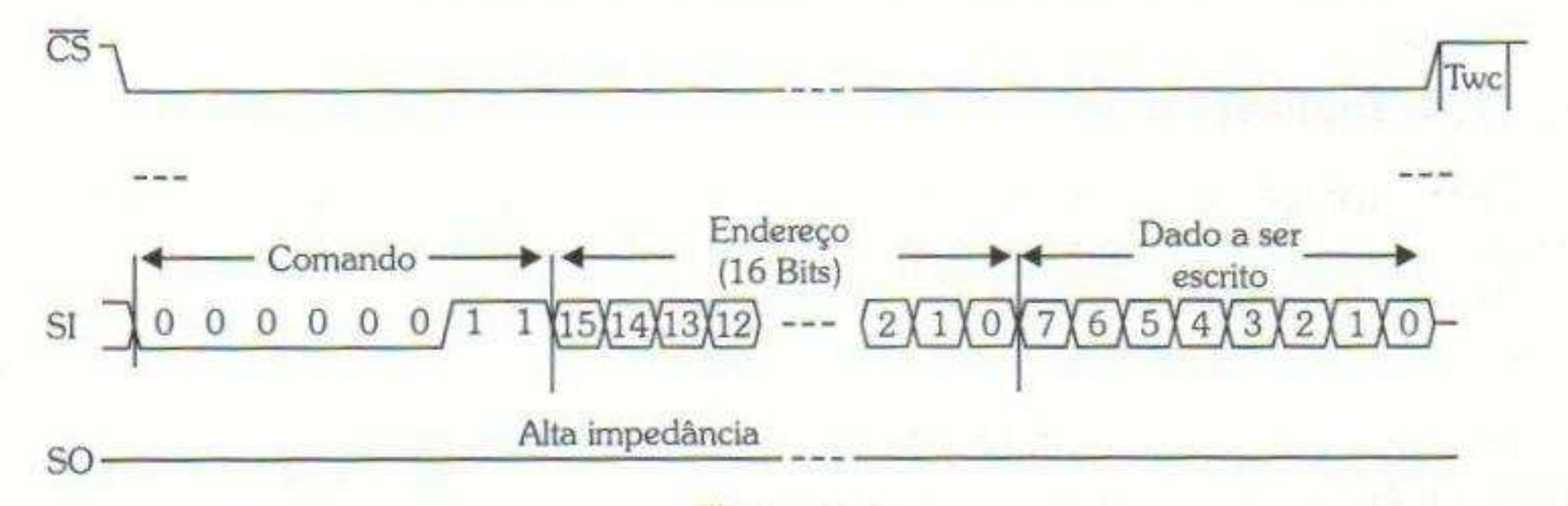

Figura 12.4

Observe que somente é possível a escrita na memória, caso ela não esteja protegida contra gravação, o que é feito por software por meio do registrador de estado, ou fisicamente, pelo pino WP da memória.

12.5.2.3. Registrador de Estado

Como já dito, as memórias SPI incluem ainda um registrador interno de estado, utilizado para controle de gravação da memória.

Bit 7	6	5	4	3	2	1	0
WPEN	x	x	x	BP1	BP0	WEL	WIP

Tabela 12.3

A função de cada bit é descrita em seguida:

- WPEN - Write Protect Enable: habilita proteção de escrita: com esse bit setado, o chip passa a estar protegido contra gravação, de acordo com os bits BP0 e BP1;
- BP1 e BP0: seleção dos segmentos de memória a serem protegidos:

BP1	BP0	Área Protegida
0	0	Nenhuma
0	1	Quarto (1/4) superior
1	0	Metade superior
1	1	Toda a memória

Tabela 12.4

- WEL - Write Enable Latch: bit de leitura que indica o estado de permissão de gravação do latch da memória;
- WIP - Write In Progress: bit de leitura que indica se a memória está em uma operação de gravação ("1") ou não ("0").

12.5.2.4. Biblioteca SPI

Neste tópico, apresentamos uma biblioteca de software adequada para realizar a comunicação com um dispositivo escravo conectado a qualquer modelo de PIC, ou mesmo outros microcontroladores (mediante pequenas modificações).

Apesar de muitos modelos de PIC possuírem interfaces de hardware SPI internas (capazes de comunicação em modo mestre e escravo), a escolha pela biblioteca de software foi feita em virtude do seu caráter didático e pela facilidade de implementação em quaisquer dispositivos.

```
/*****************************************************************/
/*  SPI.C  -  Biblioteca de comunicação SPI (mestre)             */
/*  por software                                                 */
/*                                                               */
/*  Autor : Fábio Pereira                                        */
/*****************************************************************/

// Definições dos pinos da interface SPI
// Para utilizar outros pinos, basta incluir novas definições
// no arquivo do programa onde esta biblioteca for incluída
#ifndef spi_clk
   // Definições dos pinos de comunicação
   #define spi_clk  pin_b1        // pino de clock
   #define spi_dta  pin_b0        // pino de dados
   #define spi_cs   pin_b2        // pino de seleção
#endif

// definições de comandos SPI para memória
#define spi_read_cmd     0x03
#define spi_write_cmd    0x02
#define spi_wren_cmd     0x06
#define spi_wrdi_cmd     0x04
#define spi_rdsr_cmd     0x05
#define spi_wrsr_cmd     0x01

void spi_escreve_bit (boolean bit)
```

```
// escreve um bit na interface SPI
{
   output_bit (spi_dta,bit);      // coloca o dado na saída
   output_high (spi_clk);         // ativa a linha de clock
   delay_us(1);
   output_low (spi_clk);          // desativa a linha de clock
}

void spi_escreve_byte (byte dado)
// escreve um byte na interface SPI
{
   int conta = 8;
   // envia primeiro o MSB
   while (conta)
   {
         spi_escreve_bit ((shift_left(&dado,1,0)));
         conta--;
   }
   output_float (spi_dta); // coloca a linha de dados em alta impedância
}

boolean spi_le_bit (void)
// le um bit na interface SPI
{
   boolean bit;
   output_high (spi_clk);
   delay_us (1);
   bit = input (spi_dta);
   output_low (spi_clk);
   return bit;
}

byte spi_le_byte (void)
// lê um byte na interface SPI
{
   int conta = 8, dado = 0;
   output_float (spi_dta);
   while (conta)
   {
         shift_left (&dado, 1, spi_le_bit());
         conta--;
   }
}

void spi_escreve_eeprom ( long int endereco, byte dado)
// escreve um dado em uma determinada posição da memória SPI
{
   output_low (spi_cs);                  // seleciona a memória
   spi_escreve_byte (spi_write_cmd);     // envia comando de escrita
   spi_escreve_byte (endereco >>8);      // envia endereço MSB
   spi_escreve_byte (endereco);          // envia endereço LSB
   spi_escreve_byte (dado);              // envia dado
   output_high (spi_cs);          // desativa linha CS e inicia a escrita
   // lembre-se de aguardar 5ms antes de realizar outra escrita ou leitura
}

byte spi_le_eeprom ( long int endereco )
// lê um dado de uma determinada posição da memória SPI
{
   byte dado;
   output_low (spi_cs);
```

```
   spi_escreve_byte (spi_read_cmd);      // envia comando de leitura
   spi_escreve_byte (endereco >> 8);     // envia endereço MSB
   spi_escreve_byte (endereco);          // envia endereço LSB
   dado = spi_le_byte ();                // lê o dado
   output_high (spi_cs);          // desativa linha CS e termina leitura
}

void spi_ativa_escrita_eeprom (void)
// habilita a escrita na memória
{
   output_low (spi_cs);
   spi_escreve_byte (spi_wren_cmd);
   output_high (spi_cs);
}

void spi_desativa_escrita_eeprom (void)
// desabilita a escrita na memória SPI
{
   output_low (spi_cs);
   spi_escreve_byte (spi_wrdi_cmd);
   output_high (spi_cs);
}

byte spi_le_status_eeprom (void)
// le o registrador de estado da memória
{
   byte dado;
   output_low (spi_cs);
   spi_escreve_byte (spi_rdsr_cmd);
   dado = spi_le_byte ();
   output_high (spi_cs);
}

void spi_escreve_status_eeprom (byte dado)
// escreve o dado no registrador de estado da memória SPI
{
   output_low (spi_cs);
   spi_escreve_byte (spi_wrsr_cmd);
   spi_escreve_byte (dado);
   output_high (spi_cs);
}
```

12.5.3. Protocolo I²C

O protocolo I²C (Inter Integrated Comunication - Comunicação entre Integrados) é um dos mais utilizados na comunicação de dispositivos dentro de um mesmo circuito ou equipamento eletrônico.

Esse protocolo foi desenvolvido pela Philips para facilitar o desenvolvimento de sistemas modulares para televisores e outros aparelhos eletrônicos de consumo geral.

Trata-se de um protocolo síncrono de dois fios ou linhas, sendo uma linha de clock (chamada de SCL) e outra de dados (chamada de SDA). Graças à especificação de saídas em coletor (ou dreno) aberto, o protocolo permite a

ligação de diversos dispositivos nas mesmas linhas, formando um autêntico barramento de comunicação serial, ou uma rede de dispositivos.

Atualmente o protocolo está em sua revisão número 2.1 e suporta velocidades de até 3,4 Megabits por segundo. Na realidade, a grande maioria dos dispositivos I^2C é compatível com as versões anteriores do protocolo, portanto limitadas a velocidades de 100 ou 400 Kbps, conforme o dispositivo.

A quantidade de dispositivos presente no barramento é apenas limitada pela capacitância máxima admitida que é de 400pF.

Na figura 12.5 temos um exemplo de conexão de circuitos integrados a um barramento I^2C. Repare na existência dos resistores de pull-up Rp, utilizados para manter as linhas do barramento em nível lógico "1".

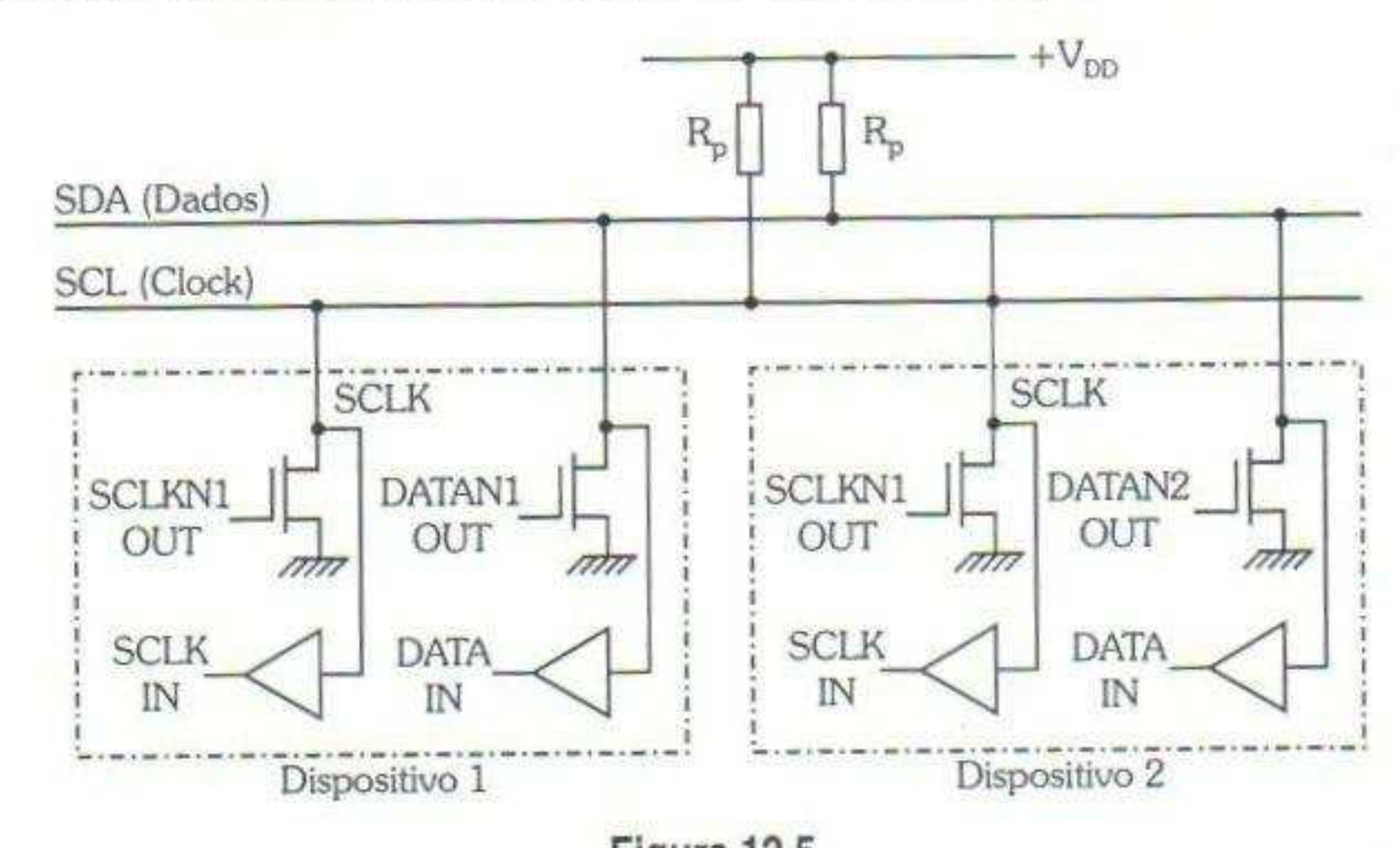

Figura 12.5

Como todo protocolo síncrono, o I^2C também é do tipo mestre-escravo, no entanto também suporta o chamado "multimastering", ou seja, a presença de diversos mestres simultaneamente no mesmo barramento. Lembre-se de que durante uma comunicação, somente um dos mestres pode estar ativo, ou ocorre uma colisão no barramento.

12.5.3.1. Características de Funcionamento

O funcionamento do protocolo baseia-se em alguns princípios:

1) A informação presente na linha de dados (SDA) somente é lida durante a fase alta da linha de clock (SCL);

2) Somente é permitido alterar o nível da linha de dados (SDA) durante a fase baixa da linha de clock (SCL);

3) Quando o barramento não está em uso, ambas as linhas permanecem desligadas (portante forçadas em nível "1" pelos resistores de pull-up).

Para sinalizar o início e o fim de uma transmissão, utiliza-se a violação da segunda regra anterior.

Assim, para sinalizar o início de transmissão (também chamado de "condição de início" ou START), o dispositivo força a linha SDA de "1" para "0", durante a fase alta do clock. Esta violação indica aos dispositivos que uma transmissão terá início.

Para sinalizar o fim de uma transmissão, é utilizada a chamada "condição de parada", ou STOP, que consiste na transição de "0" para "1" da linha SDA durante a fase alta da linha SCL.

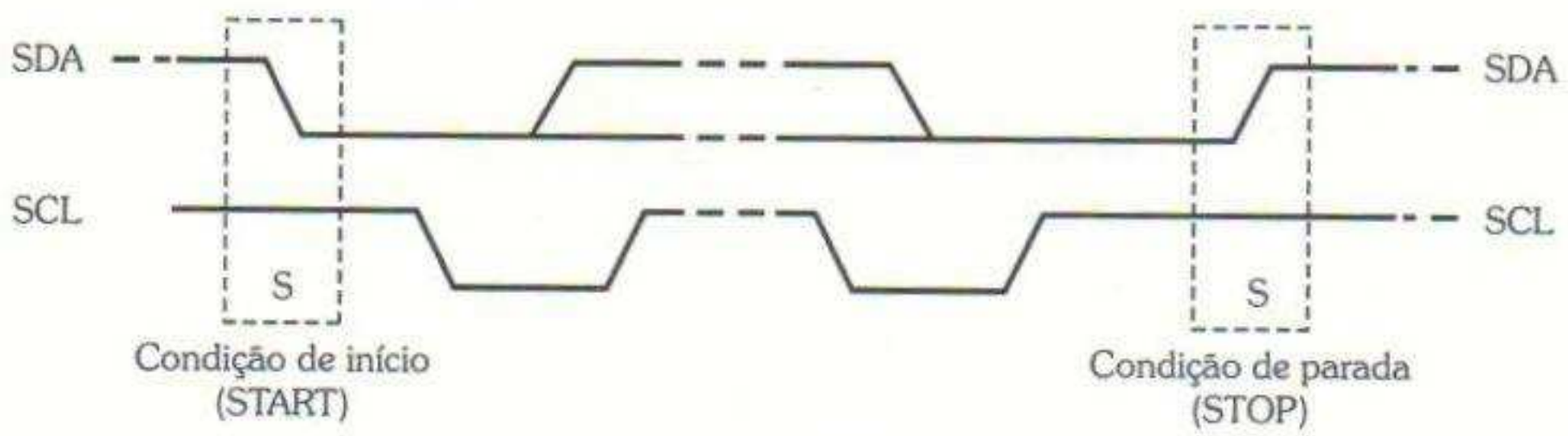

Figura 12.6

Após o bit de START, são transmitidos oito bits de dados, iniciando pelo MSB. Após o último bit (LSB) o receptor deve gerar uma condição de reconhecimento (acknowledge), o que é feito forçando a linha SDA em nível "0" antes do nono pulso de clock da linha SCL.

Caso o receptor não reconheça o dado (mantendo a linha SDA em "1" durante o nono pulso da linha SCL), o transmissor deve abortar (gerando uma condição de parada) e reiniciar a transmissão.

12.5.3.2. Formatos de Dados

Para a coexistência de diversos dispositivos em um mesmo barramento, é necessário que cada um possua identificação ou endereço próprio.

O formato básico de um comando I^2C é constituído por 7 bits de endereço, utilizados para especificar o dispositivo escravo a ser acessado, seguidos por um bit indicador de leitura/escrita, conforme a figura 12.7.

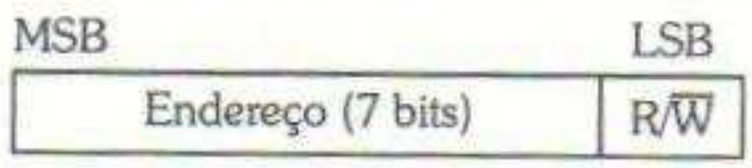

Figura 12.7

Normalmente o endereço de 7 bits é composto por duas partes: a primeira, de 4 bits, especifica o tipo de dispositivo escravo a ser acessado. A segunda, de 3 bits, especifica um entre até oito dispositivos daquele tipo, o qual será acessado.

O bit R/$\overline{W}$ indica se a operação é de leitura (1) ou escrita (0).

Um detalhe importante é que alguns valores de endereços possuem aplicação predefinida, conforme a tabela 12.5.

Endereço	R/$\overline{W}$	Significado
0000000	0	Endereço de chamada geral (broadcast)
0000000	1	Byte de início (pode ser utilizado para auxiliar dispositivos lentos a identificar o início de uma transmissão)
0000001	x	Endereço CBUS
0000010	x	Reservado para formatos diferentes de barramentos
0000011	x	Reservado para propósitos futuros
00001xx	x	Código para modo mestre de alta velocidade
0010xxx	x	Sintetizadores de voz
0011xxx	x	Interfaces de áudio PCM
0100xxx	x	Geradores de tons de áudio
0111xxx	x	Displays LED/LCD
1000xxx	x	Interfaces de vídeo
1001xxx	x	Interfaces A/D e D/A
1010xxx	x	Memórias seriais
1100xxx	x	Sintonizadores de RF
1101xxx	x	Relógios / calendários
11111xx	x	Reservado para propósitos futuros
11110xx	x	Modo de endereçamento de 10 bits

Tabela 12.5

12.5.3.3. Modo de Endereçamento de 10 Bits

O formato de uma palavra de endereçamento de 10 bits é composto de 2 bytes conforme em seguida:

S	1	1	1	1	0	A9	A8	R/$\overline{W}$	$\overline{ACK}$	A7	A6	A5	A4	A3	A2	A1	A0	$\overline{ACK}$

Figura 12.8

Em que:

- S - condição de início, **R/$\overline{\mathbf{W}}$** - bit de escrita/leitura, **$\overline{\mathbf{ACK}}$** - reconhecimento.

12.5.3.4. Chamada Geral

O endereço de chamada geral pode ser utilizado para enviar dados ou comandos a todos os dispositivos conectados ao barramento.

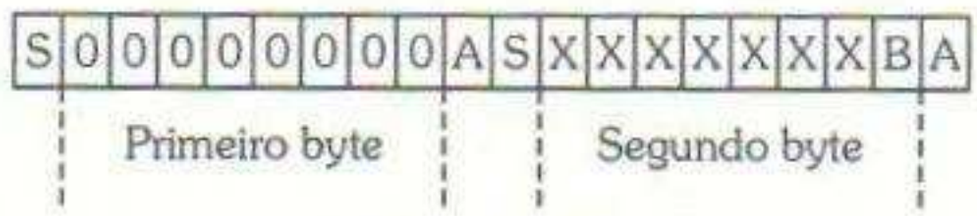

Figura 12.9

O primeiro byte do pacote contém o endereço de chamada geral, já o segundo byte pode possuir diferentes finalidades, dependendo do estado do bit B. Maiores detalhes podem ser obtidos nas especificações do protocolo.

12.5.3.5. Biblioteca de Software I²C

Em seguida temos uma biblioteca genérica para comunicação I²C em modo mestre (não suporta o modo escravo), contando inclusive com funções para leitura e gravação de memórias EEPROM seriais.

Observe que a biblioteca descrita suporta somente o funcionamento no modo mestre.

```
/*****************************************************************/
/*  I2C.C                                                        */
/*  Biblioteca I2C - Comunicação I2C por software com suporte    */
/*  a memórias EEPROM (modo mestre)                              */
/*                                                               */
/*  Autor: Fábio Pereira                                         */
/*                                                               */
/*****************************************************************/

#ifndef scl
   // Definições dos pinos de comunicação
   #define scl  pin_b1            // pino de clock
   #define sda  pin_b0            // pino de dados
   #define EEPROM_SIZE 32768  // tamanho em bytes da memória EEPROM
#endif

#define seta_scl  output_float(scl)            // seta o pino scl
#define apaga_scl  output_low(scl)             // apaga o pino scl
#define seta_sda  output_float(sda)            // seta o pino sda
#define apaga_sda  output_low(sda)             // apaga o pino sda

void I2C_start(void)
// coloca o barramento na condição de start
{
   apaga_scl;  // coloca a linha de clock em nível 0
   seta_sda;      // coloca a linha de dados em alta impedância (1)
```

```
   seta_scl;        // coloca a linha de clock em alta impedância (1)
   apaga_sda;       // coloca a linha de dados em nível 0
   apaga_scl;       // coloca a linha de clock em nível 0
}
void I2C_stop(void)
// coloca o barramento na condição de stop
{
   apaga_scl;       // coloca a linha de clock em nível 0
   apaga_sda;       // coloca a linha de dados em nível 0
   seta_scl;        // coloca a linha de clock em alta impedância (1)
   seta_sda;        // coloca a linha de dados em alta impedância (1)
}
void i2c_ack()
// coloca sinal de reconhecimento (ack) no barramento
{
   apaga_sda;       // coloca a linha de dados em nível 0
   seta_scl;        // coloca a linha de clock em alta impedância (1)
   apaga_scl;       // coloca a linha de clock em nível 0
   seta_sda;        // coloca a linha de dados em alta impedância (1)
}
void i2c_nack()
// coloca sinal de não reconhecimento (nack) no barramento
{
   seta_sda;        // coloca a linha de dados em alta impedância (1)
   seta_scl;        // coloca a linha de clock em alta impedância (1)
   apaga_scl;       // coloca a linha de clock em nível 0
}
boolean i2c_le_ack()
// efetua a leitura do sinal de ack/nack
{
   boolean estado;
   seta_sda;                // coloca a linha de dados em alta impedância (1)
   seta_scl;                // coloca a linha de clock em alta impedância (1)
   estado = input(sda);     // lê o bit (ack/nack)
   apaga_scl;               // coloca a linha de clock em nível 0
   return estado;
}
void I2C_escreve_byte(unsigned char dado)
{
// envia um byte pelo barramento I2C
   int conta=8;
   apaga_scl;               // coloca SCL em 0
   while (conta)
   {
         // envia primeiro o MSB
         if (shift_left(&dado,1,0)) seta_sda; else apaga_sda;
         // dá um pulso em scl
         seta_scl;
         conta--;
         apaga_scl;
   }
   // ativa sda
   seta_sda;
}
unsigned char I2C_le_byte()
// recebe um byte pelo barramento I2C
{
   unsigned char bytelido, conta = 8;
   bytelido = 0;
   apaga_scl;
   seta_sda;
   while (conta)
   {
         // ativa scl
         seta_scl;
         // lê o bit em sda, deslocando em bytelido
```

```
            shift_left(&bytelido,1,input(sda));
            conta--;
            // desativa scl
            apaga_scl;
        }
        return bytelido;
    }
    void escreve_eeprom(byte dispositivo, long endereco, byte dado)
    // Escreve um dado em um endereço do dispositivo
    // dispositivo - é o endereço do dispositivo escravo (0 - 7)
    // endereco - é o endereço da memória a ser escrito
    // dado - é a informação a ser armazenada
    {
        if (dispositivo>7) dispositivo = 7;
        i2c_start();
        i2c_escreve_byte(0xa0 | (dispositivo << 1)); // endereça o dispositivo
        i2c_le_ack();
        i2c_escreve_byte(endereco >> 8);     // parte alta do endereço
        i2c_le_ack();
        i2c_escreve_byte(endereco);  // parte baixa do endereço
        i2c_le_ack();
        i2c_escreve_byte(dado);                    // dado a ser escrito
        i2c_le_ack();
        i2c_stop();
        delay_ms(10); // aguarda a programação da memória
    }
    byte le_eeprom(byte dispositivo, long int endereco)
    // Lê um dado de um endereço especificado no dispositivo
    // dispositivo - é o endereço do dispositivo escravo (0 - 7)
    // endereco - é o endereço da memória a ser escrito
    {
        byte dado;
        if (dispositivo>7) dispositivo = 7;
        i2c_start();
        i2c_escreve_byte(0xa0 | (dispositivo << 1)); // endereça o dispositivo
        i2c_le_ack();
        i2c_escreve_byte((endereco >> 8));   // envia a parte alta do endereço
        i2c_le_ack();
        i2c_escreve_byte(endereco);  // envia a parte baixa do endereço
        i2c_le_ack();
        i2c_start();
        // envia comando de leitura
        i2c_escreve_byte(0xa1 | (dispositivo << 1));
        i2c_le_ack();
        dado = i2c_le_byte(); // lê o dado
        i2c_nack();
        i2c_stop();
        return dado;
    }
```

Exemplo 12.17

```
// Este exemplo pode ser executado na placa PIC APLICAÇÕES II
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)
#include <input.c>
// define os pinos de comunicação
#define scl pin_d1
#define sda pin_d0
// inclui a biblioteca I2C
#include <i2c.c>
```

```
main()
{
   byte valor, dispositivo, tecla;
   long int endereco;
   do
   {
      do
      {
         printf("\r\nLeitura ou Escrita: ");
         tecla = toupper(getc());
         putc(tecla);
      } while ( (tecla!='L') && (tecla!='E') );

      printf("\n\rDispositivo: ");
      dispositivo = gethex1();
      printf("\n\rEndereco: ");
      endereco = gethex();
      endereco = (endereco<<8)+gethex();
      if(tecla=='L') printf("\r\nValor: %X\r\n",le_eeprom( dispositivo,
endereco ) );
      if(tecla=='E')
      {
         printf("\r\nNovo valor: ");
         valor = gethex();
         printf("\n\r");
         escreve_eeprom( dispositivo, endereco, valor );
      }
   } while (TRUE);
}
```

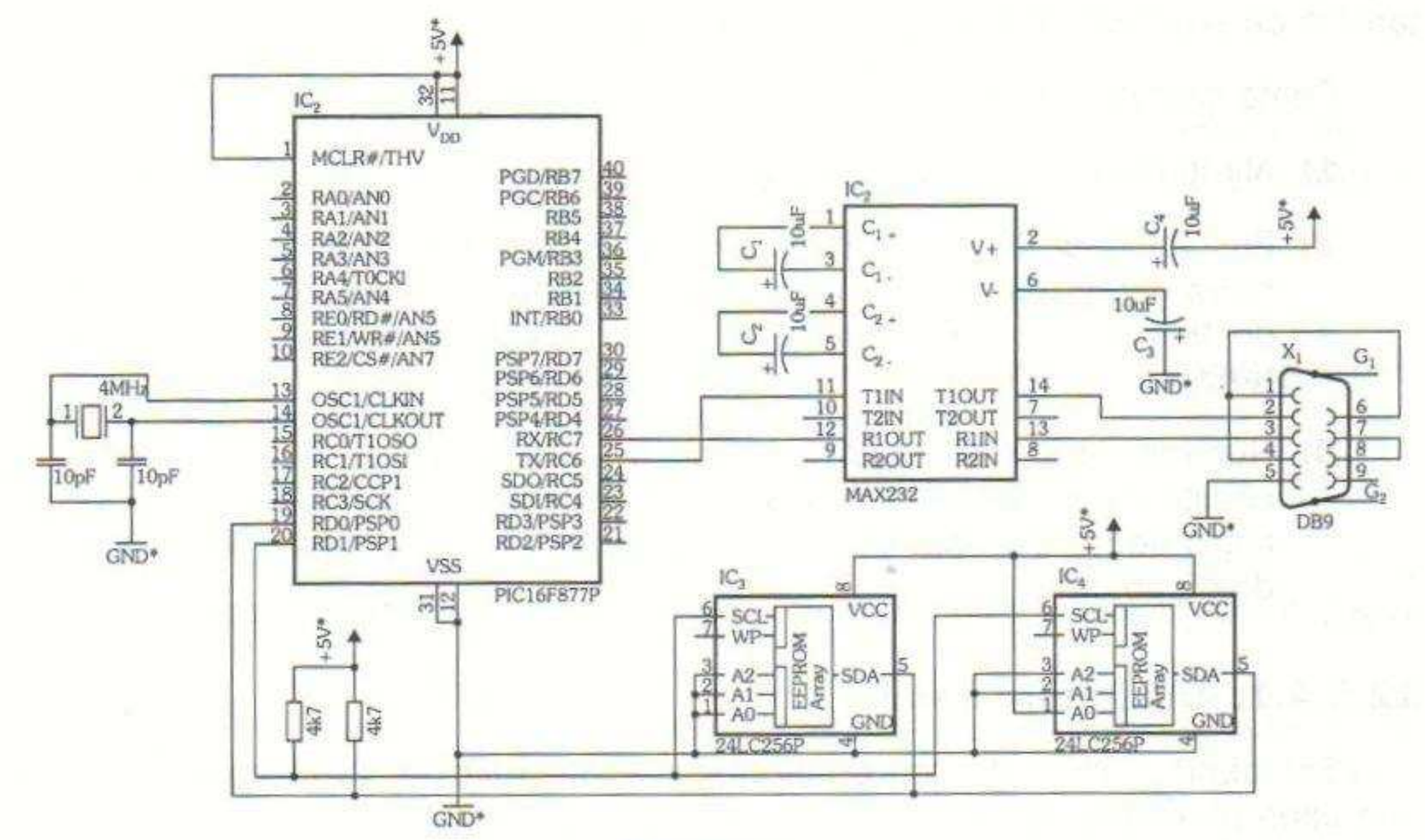

Figura 12.10

12.5.4. Protocolo 1-WIRE

O protocolo 1-Wire (1 fio) foi desenvolvido pela Dallas Semiconductor (atualmente pertencente à Maxim) para permitir a construção de dispositivos periféricos para microcontroladores e microprocessadores com um uso mínimo de recursos tanto de hardware como de software.

Existe uma grande diversidade de dispositivos compatíveis com o protocolo, tais como: chips de identificação, memórias EPROM e EEPROM seriais, relógios de tempo real, sensores de temperatura, conversores A/D, etc.

A estrutura do protocolo é baseada numa arquitetura mestre-escravo, utilizando um barramento de apenas um fio em modo half-duplex.

Eletricamente, o protocolo estabelece que os elementos de conexão ao barramento devem ser todos do tipo dreno aberto, ou seja, um dispositivo somente pode forçar o barramento em nível 0. Sendo assim, deve haver um resistor de pull-up externo de forma a garantir nível lógico 1 quando não existirem elementos forçando o barramento ao zero lógico.

O protocolo em si é do tipo assíncrono e utiliza uma arquitetura de *time--slots* ou fatias de tempo, para permitir a comunicação bidirecional half-duplex. A velocidade máxima de comunicação é de 16Kbps para o modo normal e 143 Kbps para o modo "Overdrive", o que é mais que suficiente para as pequenas tarefas de aquisição de dados e controle geral.

Como características principais do protocolo, podemos citar:

1) Alimentação entre 3 e 5,5 Volts;

2) Possibilidade de alimentação do dispositivo a partir da própria linha do barramento (modo de alimentação parasita). Neste caso, a quantidade de fios para comunicação resume-se a apenas dois: comunicação e terra;

3) Número de identificação único de 64 bits para cada dispositivo. Este código de 64 bits é composto de 8 bits para CRC, 48 bits com o número serial do dispositivo e mais 8 bits para identificação da família do dispositivo.

12.5.4.1. Arquitetura Geral

Em sendo um protocolo do tipo mestre-escravo, é desnecessário dizer que cabe ao dispositivo mestre do barramento a tarefa de iniciar e controlar a comunicação.

A inicialização do barramento é feita pelo dispositivo mestre que deve colocar o barramento em nível lógico zero por um período mínimo de 480µs. Este é o chamado pulso de reset do barramento.

Entre 15 e 60µs após o reset do barramento, os dispositivos escravos presentes aterram a linha do barramento, indicando que existem dispositivos presentes no sistema. Esse pulso de presença possui entre 60 e 240µs de duração.

Se após o pulso de reset de barramento ele continuar em nível lógico 1 por mais de 240µs, é sinal de que não há dispositivos conectados ao barramento.

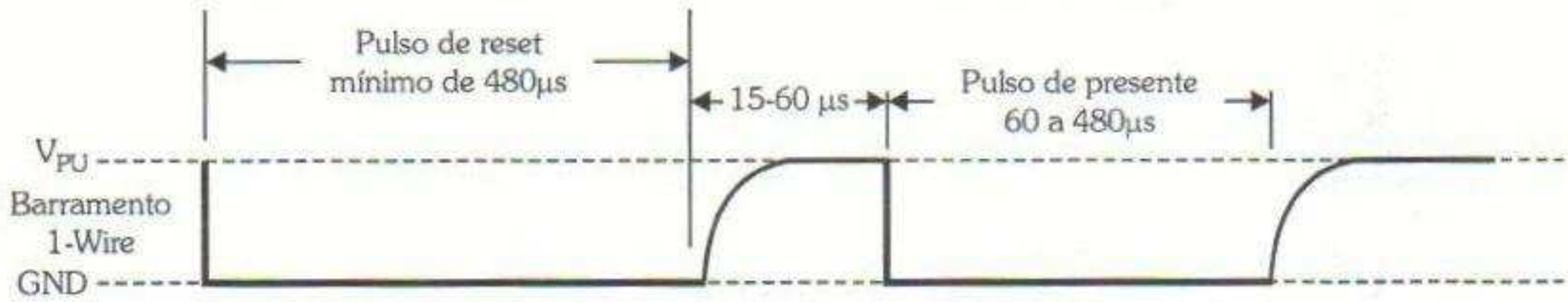

Figura 12.11

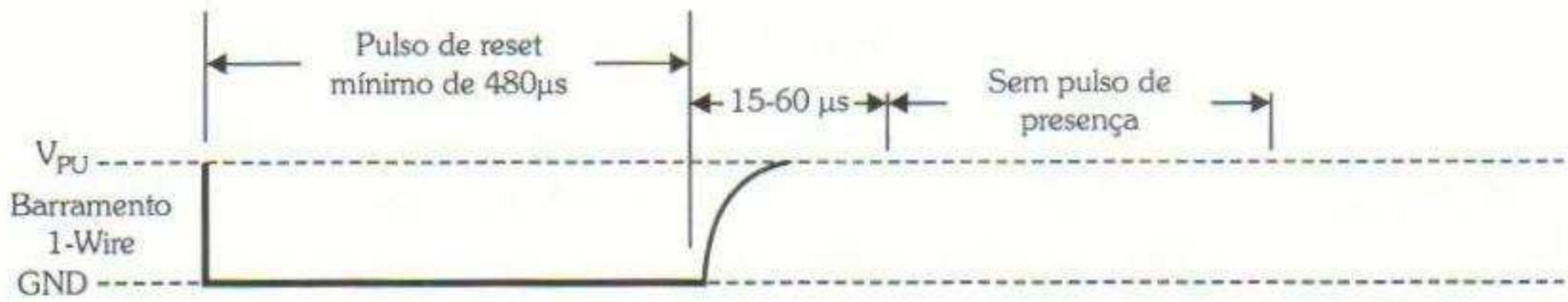

Figura 12.12

Após o reset e verificação de presença de dispositivos no barramento, o dispositivo mestre pode iniciar a comunicação com o(s) dispositivo(s) escravo(s).

Repare que existem duas situações possíveis neste caso:

1) Há apenas um dispositivo escravo no barramento (esta condição deve ser prevista no projeto do sistema);

2) Há mais de um dispositivo escravo no barramento.

No primeiro caso, o dispositivo mestre pode iniciar a comunicação com o dispositivo escravo imediatamente após o ciclo de reset / detecção de dispositivos.

No segundo caso, antes de iniciar a comunicação com os dispositivos escravos, o mestre deve providenciar a identificação individual dos dispositivos no barramento.

Doravante, consideraremos a existência de apenas um elemento escravo conectado ao barramento.

Assim, após o pulso de reset e a resposta de presença do dispositivo, o elemento mestre pode enviar comandos para o dispositivo escravo.

Os comandos são divididos em escrita e leitura. Para realizar uma escrita no barramento, o elemento mestre deve manter a linha em nível lógico "0" por no mínimo 1µs. Após este período, a linha é mantida em "0" ou desligada "nível 1", de acordo com o valor do bit a ser escrito.

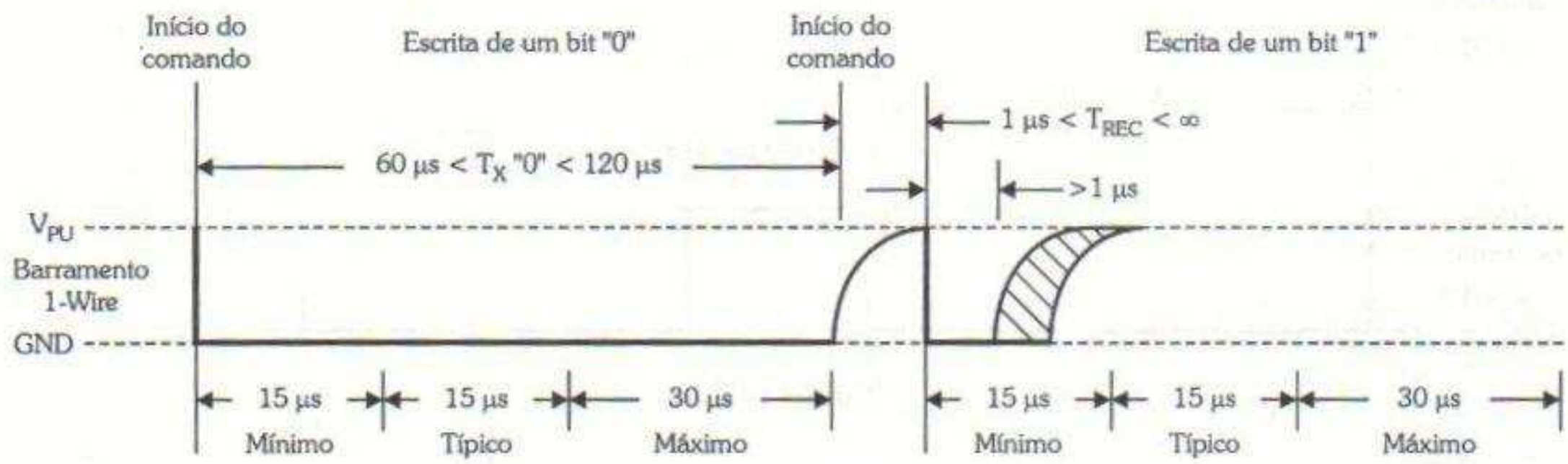

Figura 12.13

A sincronização dos elementos escravos com o dispositivo mestre é feita pela detecção da borda de descida do sinal emitido pelo mestre do barramento.

A operação de leitura de um bit é realizada de forma muito parecida com a de escrita, mas com a diferença de que o elemento mestre inicia a operação colocando a linha do barramento em nível lógico "0" por um mínimo de 1µs, em seguida desliga a saída (fazendo com que o barramento vá a nível "1" por força do resistor de pull-up externo). Aproximandamente 15µs após a borda de descida do sinal, o elemento escravo reconhece a operação e coloca o dado na linha por um período mínimo de 45µs.

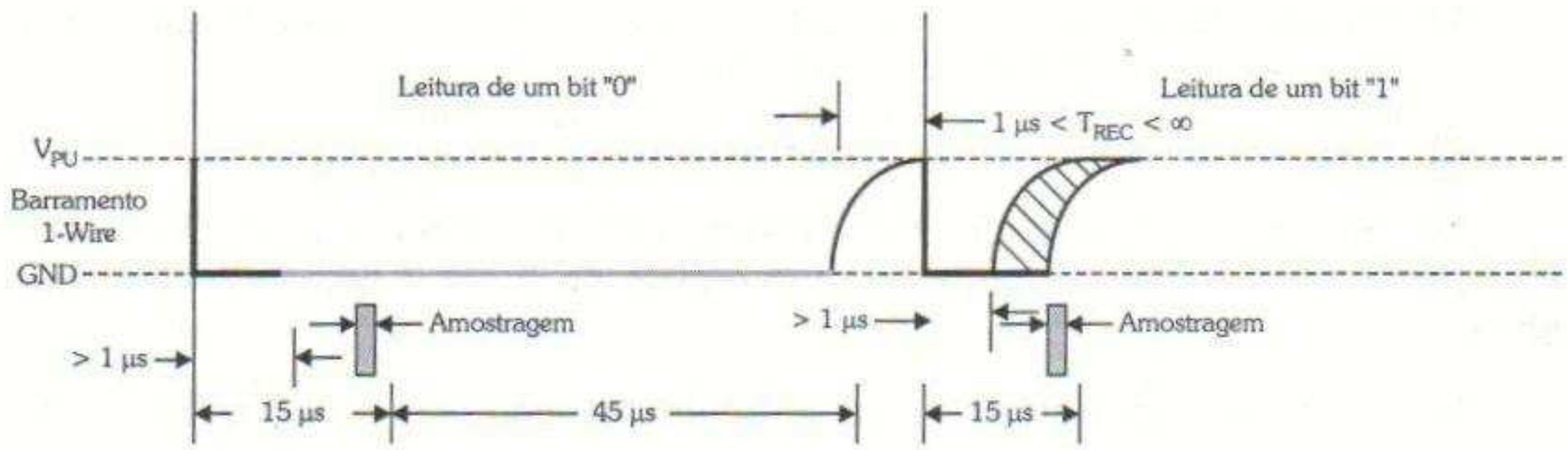

Figura 12.14

A duração dos comandos (leitura e escrita) deve estar entre 60 (mínimo) e 120µs (máximo).

Os comandos e dados que circulam pelo barramento 1-wire são sempre transmitidos iniciando pelo bit menos significativo (LSB).

Uma seqüência típica de comunicação possui no mínimo três passos:

1) Inicialização ou reset do barramento;

2) Envio de um comando de seleção de dispositivo;

3) Envio de um comando específico do dispositivo escravo.

No restante deste tópico vamos estudar o funcionamento do chip DS18S20. Vejamos algumas das suas principais características:

Figura 12.15

- Trata-se de um dispositivo de três terminais (encapsulado TO-92, como os transistores de uso geral);
- Resolução de 9 bits;
- Faixa de medição entre -55 e +125°C;
- Precisão de 0,5 graus para a faixa entre -10 e +85°C;
- Alimentação entre 3,0 e 5,5 Volts;
- Capacidade de alarme com temperatura definida pelo usuário em memória EEPROM interna;
- Pode trabalhar tanto alimentado externamente (pino VDD), como alimentado pelo próprio barramento (modo parasita).

Na tabela seguinte apresentamos um resumo dos principais comandos encontrados no chip DS18S20:

Nome	Código	Descrição
SearchROM	0xF0	Este comando fará com que os dispositivos presentes no barramento passem a transmitir seqüencialmente cada bit do seu endereço de 64 bits interno. Cada bit é transmitido duas vezes: primeiro o valor real do bit e em seguida o valor invertido. Pela análise do par de bits o mestre pode determinar se há mais de um elemento no barramento cujos bits de endereço coincidem ou no caso negativo, o valor do bit. Para cada par de bits, o dispositivo mestre deve enviar um bit de forma a especificar qual dos elementos continuará a receber e enviar os próximos bits.
ReadROM	0x33	Após receber este comando, o dispositivo escravo passa a transmitir serialemente o seu endereço de 64 bits. Este comando somente deve ser utilizado quando há apenas um dispositivo no barramento; caso contrário, haverá colisões e o endereço não será válido.

Tabela 12.6 (Continua)

Nome	Código	Descrição
MatchROM	0x55	O dispositivo mestre envia este comando seguido por 64 bits que identificam o dispositivo escravo desejado. Após este comando, somente o dispositivo endereçado pode receber dados ou comandos até que um reset seja efetuado.
SkipROM	0xCC	Este comando permite que o dispositivo mestre envie um comando para todos os dispositivos presentes no barramento, sem a necessidade de endereçar cada dispositivo. Este comando também deve ser utilizado para permitir a comunicação com apenas um dispositivo 1-wire.
Alarm Search	0xEC	Este comando funciona de forma parecida com o comando SearchROM, com a diferença de que somente os dispositivos com o flag de alarme setado responderão com os seus endereços.
Convert T	0x44	Inicia a conversão de temperatura no(s) dispositivo(s) selecionado(s). Observe que no caso do modo de alimentação parasitária, o dispositivo mestre deve providenciar uma alimentação positiva no barramento (além do resistor de pull-up) até 10µs após este comando, de forma a garantir o fornecimento de corrente suficiente para completar o ciclo de conversão. Esse tempo pode chegar a até 750ms.
Write ScratchPad	0x4E	Permite a escrita de dois bytes na memória de rascunho (bytes 2 e 3).
Read ScratchPad	0xBE	Leitura da memória de rascunho. Após este comando, o dispositivo escravo envia 9 bytes com o conteúdo da memória de rascunho do chip, seguido por um valor de CRC de 8 bits.
Copy ScratchPad	0x48	Copia o conteúdo das posições 2 e 3 da memória de rascunho (registradores TH e TL) para a EEPROM interna do CHIP. Observe que no caso do modo de alimentação parasitária, o dispositivo mestre deve providenciar uma alimentação positiva no barramento (além do resistor de pull-up) até 10µs após este comando, de forma a garantir o fornecimento de corrente suficiente para completar o ciclo de programação da memória. Esse ciclo pode durar até 10ms.
Recall E^2	0xB8	Copia os valores de TH e TL da EEPROM para a memória de rascunho. Durante a leitura da EEPROM, o dispositivo mestre pode realizar leituras de bit para verificar se o comando foi terminado ou não. Caso a leitura seja completada, o bit lido é de nível "1".
Read Power Supply	0xB4	Verifica o tipo de alimentação utilizado pelo dispositivo escravo. Após este comando, o mestre deve realizar uma leitura de bit. Os dispositivos alimentados externamente não alteram o barramento, já os dispositivos no modo parasita aterram o barramento. Assim, se o bit lido pelo mestre for "0", indica que o dispositivo está alimentado no modo parasita, caso o bit seja "1", utiliza alimentação convencional.

Tabela 12.6

Na tabela adiante temos os endereços e conteúdo de cada posição da memória de rascunho do chip:

Endereço	Conteúdo	Descrição
0	Temp LSB	Parte menos significativa da temperatura medida
1	Temp MSB	Parte mais significatica da temperatura medida
2	TH	Parte alta do alarme de temperatura
3	TL	Parte baixa do alarme de temperatura
4	Res	Reservado (lido como 0xFF)
5	Res	Reservado (lido como 0xFF)
6	Count Remain	Contagem faltante
7	Count per °C	Contagem por °C é sempre 16 (0x10)

Tabela 12.7

Os registradores de valor da temperatura (Temp) possuem o seguinte formato interno:

Registrador	bit 7	6	5	4	3	2	1	0
Temp LSB	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}
Temp MSB	S	S	S	S	S	S	S	S

Tabela 12.8

A temperatura medida pode ser facilmente lida, já que cada bit dos registradores vale 0,5 °C. A temperatura máxima (+125 °C) será lida como 0x00FA e a temperatura mínima (-55 °C) será lida como 0xFF92.

É possível ainda realizar uma leitura mais precisa, utilizando a seguinte fórmula:

$$\text{Temperatura} = \text{Temp} - 0.25 + \frac{\text{count_per_c} - \text{count_remain}}{\text{count_per_c}}$$

12.5.4.2. Biblioteca 1-Wire

Em seguida temos então a listagem da biblioteca de rotinas para comunicação utilizando o protocolo 1-Wire:

```
/*****************************************************************/
/*  1WIRE.C - Biblioteca de comunicação 1-wire                   */
/*                                                               */
/*  Autor: Fábio Pereira                                         */
/*                                                               */
/*****************************************************************/
```

```
// definição do pino de comunicação
// Para utilizar outro pino, basta incluir uma nova definição antes
// de incluir este arquivo na aplicação desejada
#ifndef pino_1w
   #define pino_1w pin_d0  // define o pino RD0 como comunicação 1-wire
#endif

#inline
void seta_saida_1w (void)
// coloca a saída em la impedância
{
   output_float (pino_1w);
}

#inline
void limpa_saida_1w (void)
// coloca a saída em nível 0
{
   output_low (pino_1w);
}

boolean reset_1w(void)
// reseta os dispositivos no barramento
{
   boolean presente;
   limpa_saida_1w ();
   delay_us(480);
   seta_saida_1w ();
   delay_us(60);
   presente = input (pino_1w);
   delay_us (240);
   return (presente);
   // 0 = dispositivo presente
   // 1 = nenhum dispositivo detectado
}

void alimenta_barramento_1w (void)
// força o barramento em nível alto
// utilizado com dispositivos alimentados no modo parasita
{
   output_high (pino_1w);
   delay_ms(1000);
   seta_saida_1w();
}

boolean le_bit_1w (void)
// lê um bit do barramento 1-wire
{
   // dá um pulso na linha, inicia quadro de leitura
   limpa_saida_1w ();            // coloca saida em zero
   seta_saida_1w ();             // retorna a saída a um
   delay_us (15);                // aguarda o dispositivo colocar
                                 // o dado na saída
   return (input(pino_1w));      // retorna o dado
}

void escreve_bit_1w (boolean bit)
// escreve um bit no barramento 1-wire
{
   limpa_saida_1w ();
   if (bit) seta_saida_1w (); // coloca dado 1 na saida
```

```
   delay_us (120);
   seta_saida_1w ();
}

byte le_byte_1w (void)
// lê um byte do barramento 1-wire
{
   byte i, dado = 0;
   // lê oito bits iniciando pelo bit menos significativo
   for (i=0;i<8;i++)
   {
         if (le_bit_1w ()) dado|=0x01<<i;
         delay_us(90);  // aguarda o fim do quadro de leitura
                        // do bit atual
   }
   return (dado);
}

void escreve_byte_1w (char dado)
// escreve um byte no barramento 1-wire
{
   byte i, temp;
   // envia o byte iniciando do bit menos significativo
   for (i=0; i<8; i++)
   {
         temp = dado>>i;          // desloca o dado 1 bit à direita
         temp &= 0x01;            // isola o bit 0 (LSB)
         escreve_bit_1w (temp);// escreve o bit no barramento
   }
}
```

Em seguida temos um programa que efetua a comunicação com um sensor digital de temperatura DS 18S20 por meio do pino RD0 do PIC 16F877, ou outro que se desejar utilizar.

Exemplo 12.18

```
// Este exemplo pode ser executado na placa PIC APLICAÇÕES II
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)

#include <1wire.c>

int calc_crc(int *dados, int quantidade)
{
   int shift_reg=0, data_bit, sr_lsb, fb_bit, i, j;

   for (i=0; i<quantidade; i++)  // loop de bytes
   {
      for(j=0; j<8; j++) // loop de bits
      {
         data_bit = (dados[i]>>j)&0x01;
         sr_lsb = shift_reg & 0x01;
         fb_bit = (data_bit ^ sr_lsb) & 0x01;
         shift_reg = shift_reg >> 1;
         if (fb_bit)
```

```
            {
                shift_reg = shift_reg ^ 0x8c;
            }
        }
    }
    return(shift_reg);
}

void main(void)
{
    int buffer[9], conta;
    float temp;
    while (true)
    {
            reset_1w ();
            escreve_byte_1w (0xcc);        // comando skip ROM
            escreve_byte_1w (0x44);        // inicia conversão da temperatura
            reset_1w ();                   // reseta o dispositivo
            escreve_byte_1w (0xcc);        // comando skip ROM
            // comando de leitura da memória de rascunho
            escreve_byte_1w (0xbe);
            // efetua a leitura dos nove bytes da memória de rascunho
            for (conta = 0; conta<9; conta++)
                    buffer[conta]=le_byte_1w ();
            printf ("Temp LSB = %u\r\n",buffer[0]);
            printf ("Temp MSB = %u\r\n",buffer[1]);
            printf ("TH = %u\r\n",buffer[2]);
            printf ("TL = %u\r\n",buffer[3]);
            printf ("Contagem Remanescente = %u\r\n",buffer[6]);
            printf ("Contagem por grau C = %u\r\n",buffer[7]);
            printf ("CRC = %u\r\n",buffer[8]);
            printf ("CRC calculado = %u\r\n", calc_crc (buffer,8));
            reset_1w ();
            escreve_byte_1w (0xcc);
            escreve_byte_1w (0xb4);
            if (le_bit_1w()) printf ("Modo alimentado\r\n"); else printf
("Modo parasita\r\n");
            printf ("Temperatura = %Ld",((long)(buffer[1]<<8) + bu-
ffer[0])>>1);
            if (bit_test(buffer[0],0)) printf (".5");
            printf (" graus celsius (menos preciso)\r\n");
            temp = (long) (buffer[1]<<8) + buffer[0];
            temp = (temp / 2) - 0.25 + (float) ((buffer[7]-
buffer[6])/buffer[7]);
            printf ("Temperatura = %3.2f graus celsius\r\n", temp);
            printf ("Proximo ?\r\n");
            while (getc()!=13);
    }
}
```

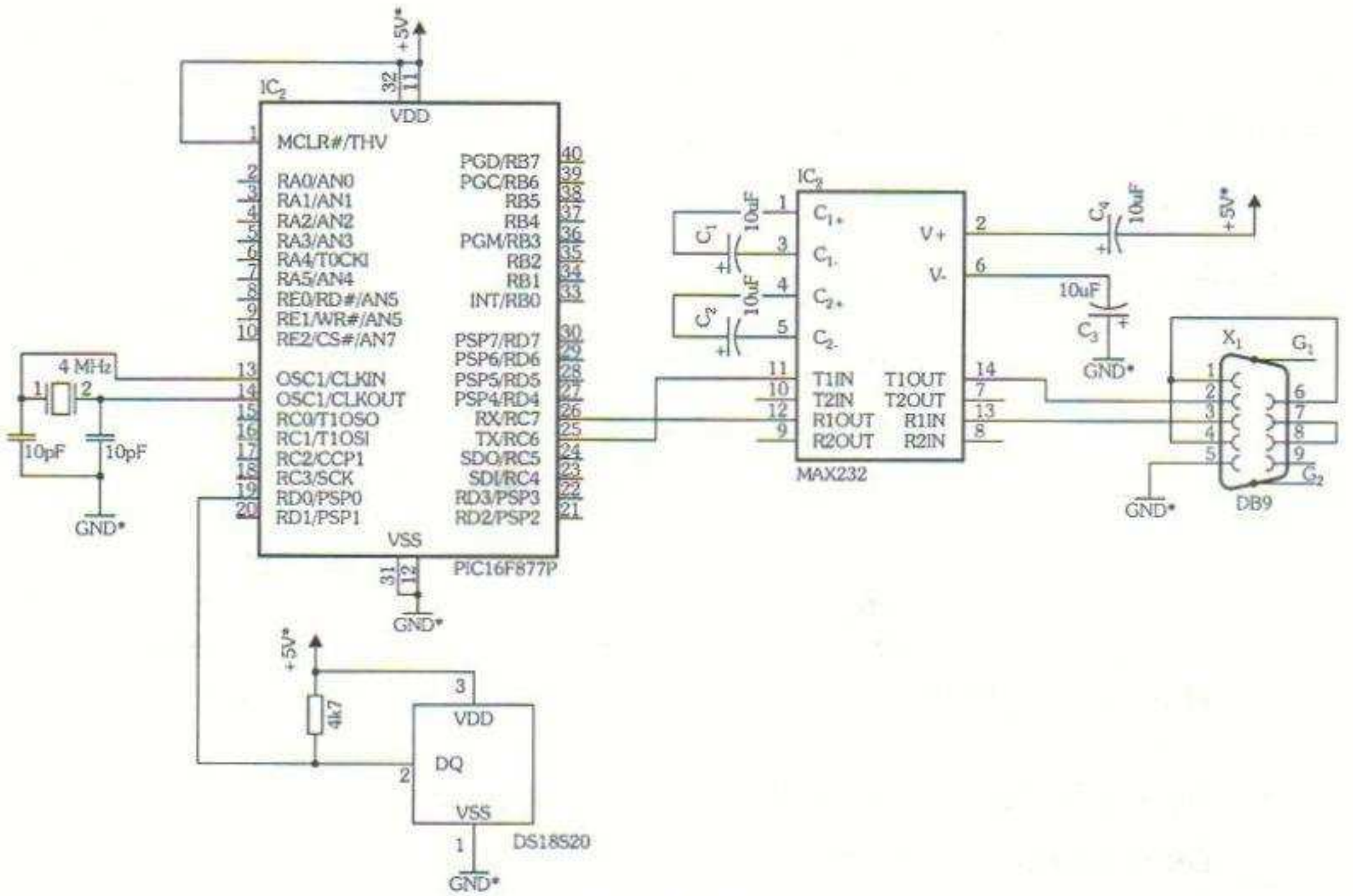

Figura 12.16

12.5.5. LIN

O protocolo LIN (Local Interconnect Network - Rede de Interconexão Local) é fruto de estudos e especificações emitidas por alguns dos maiores fabricantes da indústria automobilística mundial, como Audi, BMW, DaimlerChrysler, Volkswagen e Volvo, além de outras empresas como Motorola e Volcano Communicatons Technologies.

Trata-se de um protocolo de barramento simples, de baixa velocidade (até 20Kbps) e baixo custo, utilizando apenas um fio para conexão entre os elementos de comunicação em uma arquitetura mestre-escravo.

A idéia básica por trás do LIN era desenvolver um protocolo simples, de forma a viabilizar a utilização de USARTs e SCIs existentes nos MCUs disponíveis no mercado, ou ainda utilizando implementações por software, tornando-o assim um protocolo de muito baixo custo.

Apesar de aparentemente ser um concorrente do CAN, este fato não é de todo verdadeiro, já que CAN é um protocolo de velocidade muito mais elevada, adequado a tarefas às quais o LIN simplesmente não possui velocidade suficiente para atender.

Por outro lado, o protocolo LIN permite atender com custo menor a algumas aplicações nas quais o CAN seria de implementação inviável, seja por questões técnicas ou por questões econômicas.

Vejamos algumas características desse protocolo:

- Ausência de arbitragem de barramento, já que é permitida a existência de somente um dispositivo mestre, além de qualquer quantidade de dispositivos escravos;
- Garantia de tempos de latência para transmissão de sinais;
- Largura de quadro da mensagem selecionável em 2,4 ou 8 bytes;
- Flexibilidade de configuração;
- Possibilidade de endereçamento de múltiplos escravos simultaneamente com sincronização de tempo, eliminando a necessidade de cristais de quartzo ou ressonadores cerâmicos nos dispositivos escravos;
- Detecção de erros e verificação de dados por checksum;
- Detecção de nós de rede com defeito;
- Custo mínimo para os componentes semicondutores;
- Capacidade de entrar em modo de baixo consumo de energia e de despertar ao comando do mestre.

Para permitir a conexão a um barramento LIN, o dispositivo de comunicação deve apresentar uma saída capaz de atuar na condição de coletor ou dreno aberto, tal como descreve a figura seguinte:

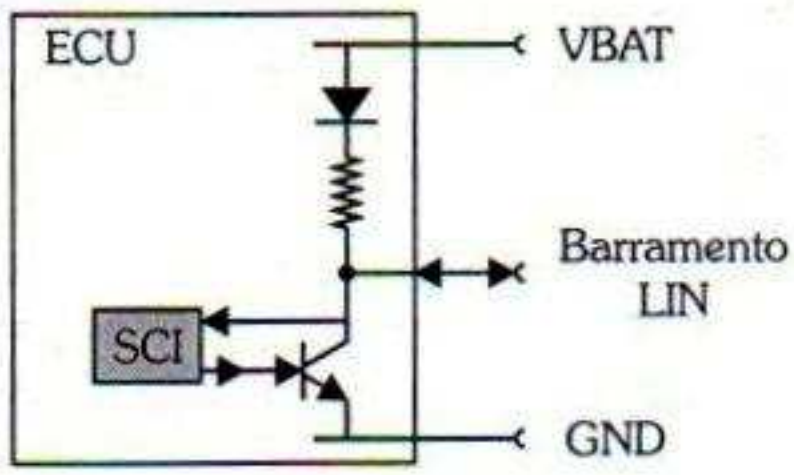

Figura 12.17

Um quadro de mensagem LIN é composto de duas partes: um cabeçalho e uma resposta.

O cabeçalho é transmitido pelo dispositivo mestre e é composto de três partes: um campo de sincronização e sinalização de início de transmissão, um campo de sincronismo e um campo identificador.

A resposta pode ser tanto uma mensagem do mestre para um dispositivo escravo, quanto uma resposta do escravo para o mestre. É composta de no

mínimo dois e no máximo oito campos de dados, seguidos por um campo de checagem (checksum).

Uma mensagem LIN possui o formato típico seguinte:

Figura 12.18

Vejamos agora cada elemento da mensagem em maiores detalhes:

12.5.5.1. Campo SYNCH BREAK

É utilizado para o propósito de sinalização de início de transmissão e para sincronização dos elementos escravos com o dispositivo mestre do barramento.

A especificação do protocolo determina que a fase baixa (nível 0, também chamado de estado dominante) do campo SYNCH BREAK possua duração mínima de 13 tempos de bit do mestre.

Já a fase alta (nível 1, também chamado de estado recessivo) é utilizada para permitir a detecção do bit de start do campo seguinte

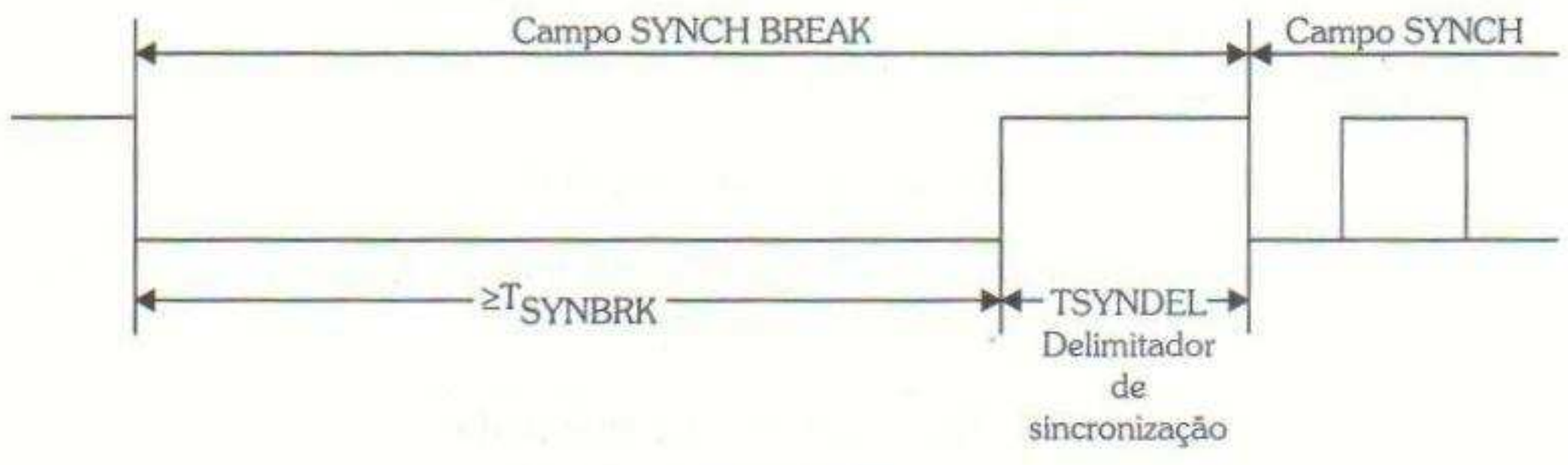

Figura 12.19

12.5.5.2. Campo SYNCH

É utilizado para fins de sincronização de clock sendo composto de um típico quadro de transmissão de USART: 1 bit de início (START), oito bits de dados (no caso, 0x55) e 1 bit de parada.

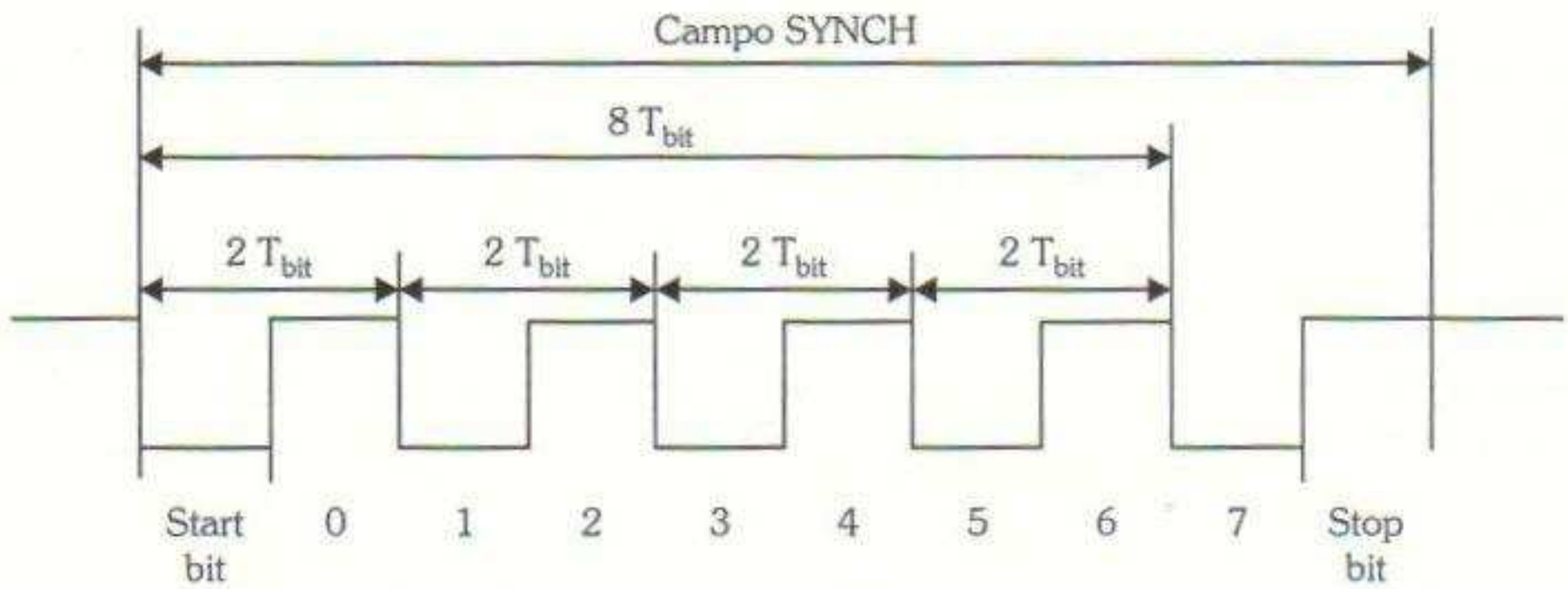

Figura 12.20

A tarefa de sincronização e extração de base de tempo é feita pela medição do período entre cada borda de descida do sinal, podendo ser medido o tempo de 2, 4, 6 ou 8 bits. Esta tarefa deve ser realizada por uma rotina de software ou com auxílio de timers internos.

12.5.5.3. Campo Identificador

É utilizado para identificar o tipo de mensagem e também o tamanho dela.

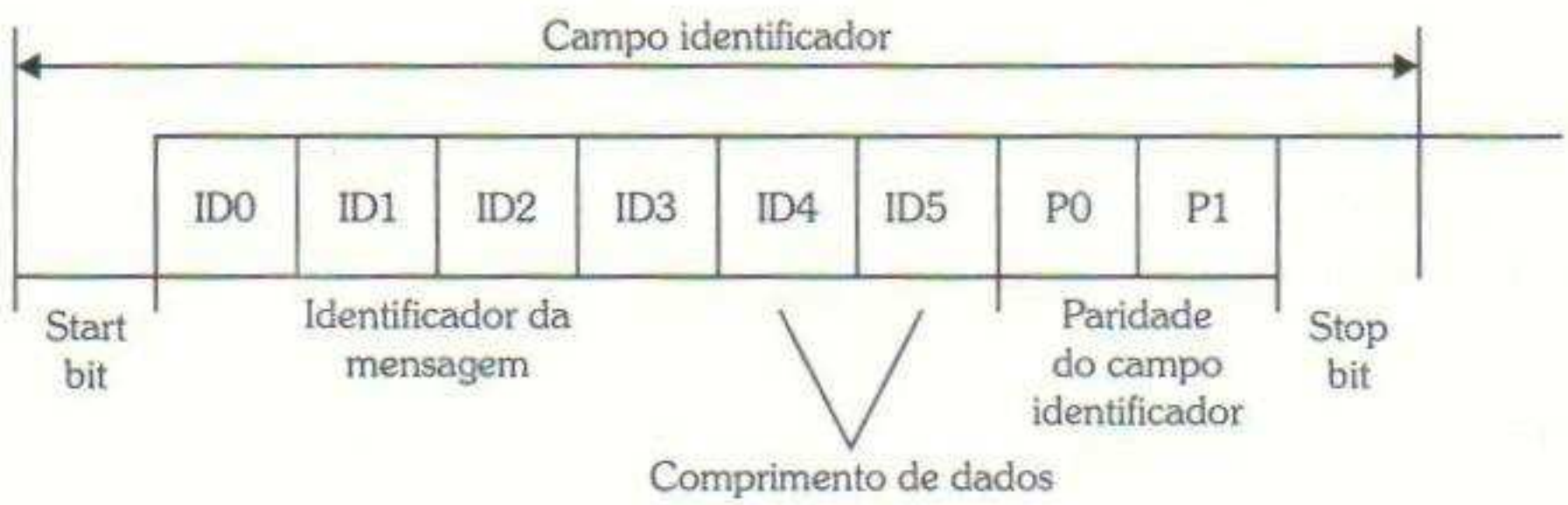

Figura 12.21

Os bits ID0, ID1, ID2 e ID3 servem para identificar o tipo de mensagem.

Os bits ID4 e ID5 especificam o número de bytes de dados na resposta, conforme a tabela:

ID5	ID4	Número de campos de dados de resposta
0	0	2
0	1	2
1	0	4
1	1	8

Tabela 12.9

Os bit P0 e P1 são utilizados para calcular a paridade:

$$P0 = ID0 \oplus ID1 \otimes ID2 \oplus ID4$$

$$P1 = \overline{ID1 \oplus ID3 \otimes ID4 \oplus ID5}$$

Os identificadores 0x3C, 0x3D, 0x3E e 0x3F, bem como os campos de identificador 0x3C, 0x7D, 0xFE e 0xBF, são reservados para comandos especiais do protocolo.

12.5.5.4. Campo de dados

Os campos de dados são encontrados na área de resposta do quadro de mensagem.

Cada campo de dados consiste em 10 bits, seguindo o mesmo formato utilizado nas USARTs dos microcontroladores, ou seja: primeiro um bit de início (START), seguido de 8 bits de dados (iniciando pelo LSB) e em seguida um bit de parada (STOP).

Figura 12.22

12.5.5.5. Campo de Checagem

O campo de checagem pode ser utilizado pelo dispositivo que recebe o pacote de resposta, para verificar a integridade da informação contida nos campos de dados.

Para calcular o "checksum", utiliza-se o resultado invertido da soma módulo 256. Para tanto soma-se o byte de cada campo de dados ao carry da soma anterior e o resultado final é negado (ou invertido).

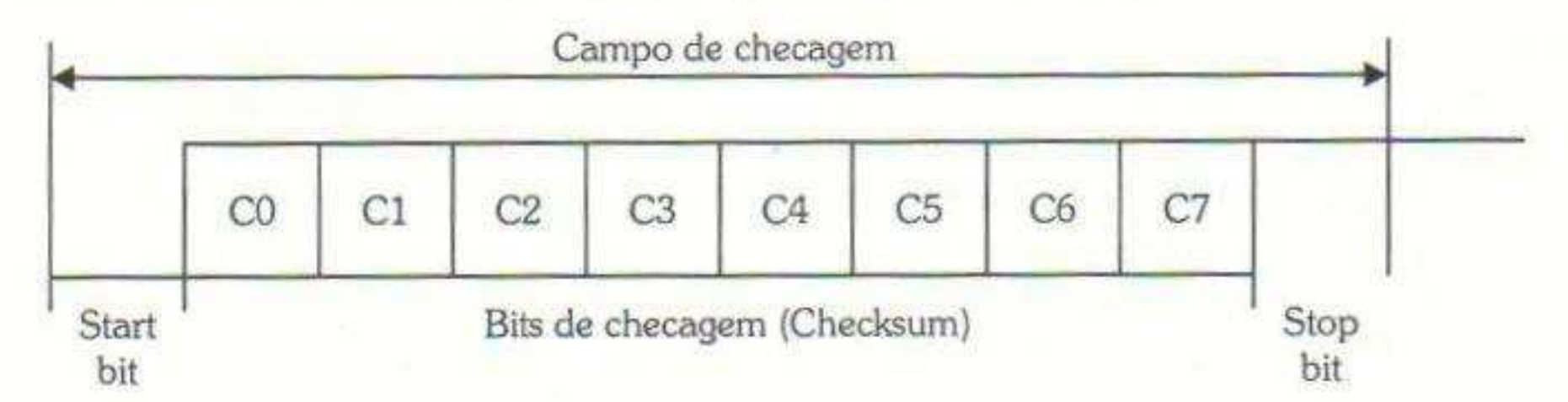

Figura 12.23

O valor do "checksum" é transmitido iniciando pelo bit LSB.

12.5.5.6. Identificadores Reservados

Como já dito, alguns identificadores são reservados para uso interno do protocolo LIN, conforme a tabela 12.10:

Identificador	Descrição
0x3C	Mensagem de requisição do mestre: utilizado para enviar comandos e dados de um dispositivo mestre para um dispositivo escravo.
0x7D	Mensagem de recepção do escravo: utilizado para solicitar a um dispositivo escravo endereçado previamente, a transmissão de dados para o dispositivo mestre.
0xFE	Mensagem estendida para aplicações definidas pelo usuário.
0xBF	Mensagem reservada para extensões futuras do protocolo LIN.

Tabela 12.10

12.5.6. Protocolo CAN

O protocolo CAN (Controller Area Network - Rede de Área de Controladores) foi desenvolvido pela BOSCH para permitir a comunicação de alta velocidade (1 Mbps) e de forma confiável, tendo sido desenvolvido inicialmente para aplicações automotivas relacionadas principalmente aos dispositivos de ignição e injeção eletrônicas, além de sensoriamento automotivo.

O protocolo original da BOSCH não especifica a mídia de transmissão a ser utilizada, no entanto a Organização Internacional de Padrões (ISO) e a Sociedade de Engenheiros Automotivos (SAE) definiram posteriormente os padrões elétricos de transmissão ISO 11898 para altas velocidades, ISO 11519 para baixas velocidades e J1939 para barramentos de caminhões e ônibus. Todos os protocolos elétricos utilizam interfaces diferenciais ao estilo 485.

A camada de enlace físico do protocolo utiliza uma arquitetura CSMA/CD (Carrier Sense Multiple Access / Colision Detect - Acesso Múltiplo por detecção de portadora e detecção de colisão) tal qual utilizada em redes locais Ethernet.

A arquitetura CSMA/CD permite que qualquer dispositivo conectado ao barramento possa ocupar o barramento, bastando verificar antes a existência ou não de ocupação do barramento.

Ainda em relação ao nível mais baixo do protocolo, os estados da linha são definidos como dominante (nível lógico "0") e recessivo (nível lógico "1").

Partindo para as camadas mais altas do protocolo, podemos dizer que CAN é baseado em uma arquitetura de mensagens em que todos os elementos conectados ao barramento recebem as mensagens circulantes nele e somente o nó de destino interpreta e reconhece a mensagem, emitindo uma mensagem de reconhecimento.

Outra característica interessante do protocolo é a capacidade de um nó solicitar informações de outro nó, o que é chamado RTR (Remote Transmit Request - Requisição de Transmissão Remota).

Existem basicamente quatro tipos de mensagem no protocolo: mensagens de dados, mensagens remotas (mensagens de dados com o bit RTR setado), mensagens de erro e mensagens de sobrecarga.

Devido à grande complexidade do protocolo, a implementação dele por software na maioria dos casos é inviável, restando somente a opção de interfaces CAN por hardware.

Existem diversos chips com capacidade de comunicação CAN no mercado. Como exemplo, podemos citar os novos PICs da série 18F (18Fxx8), dotados de interface CAN interna, além de CIs dedicados para o interfaceamento CAN geral como os CIs MCP 250xx e o transceptor MCP 2551, todos da Microchip.

12.5.7. Interfaces Elétricas

A grande maioria dos protocolos de comunicação vistos até aqui (com exceção do SPI, I^2C e 1-Wire) não especifica diretamente o padrão elétrico do sinal a ser transmitido / recebido.

Por isso os órgãos internacionais de padronização, tais como ANSI, EIA e ITU, desenvolveram protocolos ou padrões elétricos de comunicação, que foram e continuam sendo adotados para as tarefas de comunicação de dados.

Vejamos então as duas principais interfaces elétricas aplicáveis aos protocolos deste livro.

12.5.7.1. Interface TIA/EIA-232

A interface elétrica TIA/EIA-232 foi padronizada pela primeira vez em 1962, pelo EIA, para facilitar e padronizar o projeto de equipamentos terminais de dados (DTEs) e equipamentos de comunicação de dados (DCEs).

Com o decorrer do tempo, o padrão veio sendo modificado e atualmente encontra-se na sua revisão F, que é compatível com as normas V.24, V.28 e ISO 2110.

O protocolo TIA/EIA-232 especifica:

- Padrão elétrico do sinal: as características de tensão, impedância da linha e taxa de variação são especificadas da seguinte forma:
  - Sinal de saída: de +5 a +15V para o bit 0 e de -5 a -15V para o nível 1;
  - Sinal de entrada: de +3 a +15V para o bit 0 e de -3 a -15V para o nível 1;
  - Características de impedância e carga da linha: o padrão especifica que o transmissor deve possuir uma impedância de saída maior que 300 Ohms, o receptor deve possuir uma impedância de entrada entre 3 e 5k Ohms e a capacitância de carga não deve ultrapassar os 2,5 nF. O comprimento máximo do cabo, neste caso, fica limitado a aproximadamente 20 metros para cabos blindados e a 40 metros para cabos simples, isto porque os cabos blindados apresentam uma capacitância mútua maior que os cabos comuns;
  - Taxa de variação: o protocolo especifica um "slew rate" máximo de 30 V/µs, o que limita a velocidade de comunicação a aproximadamente 200 Kbps, apesar de este limite ser comumente ultrapassado.
- Conectores padronizados: o protocolo especifica conectores de 25 pinos do tipo "D" (chamados de DB25). O conector do DCE deve possuir pinagem do tipo fêmea e do DTE, pinagem do tipo macho. Observe que o conector de 25 pinos cedeu, ao longo tempo, lugar para os conectores de 9 pinos (chamados de DB9), sendo estes últimos os mais utilizados atualmente. A pinagem do conector DB9 pode ser vista na figura 12.24.
- Linhas de "Handshake": o protocolo TIA/EIA-232 especifica ainda algumas linhas adicionais, além das de comunicação de dados propriamente ditas, para finalidades de controle da comunicação e monitoração de estado do DTE e DCE.

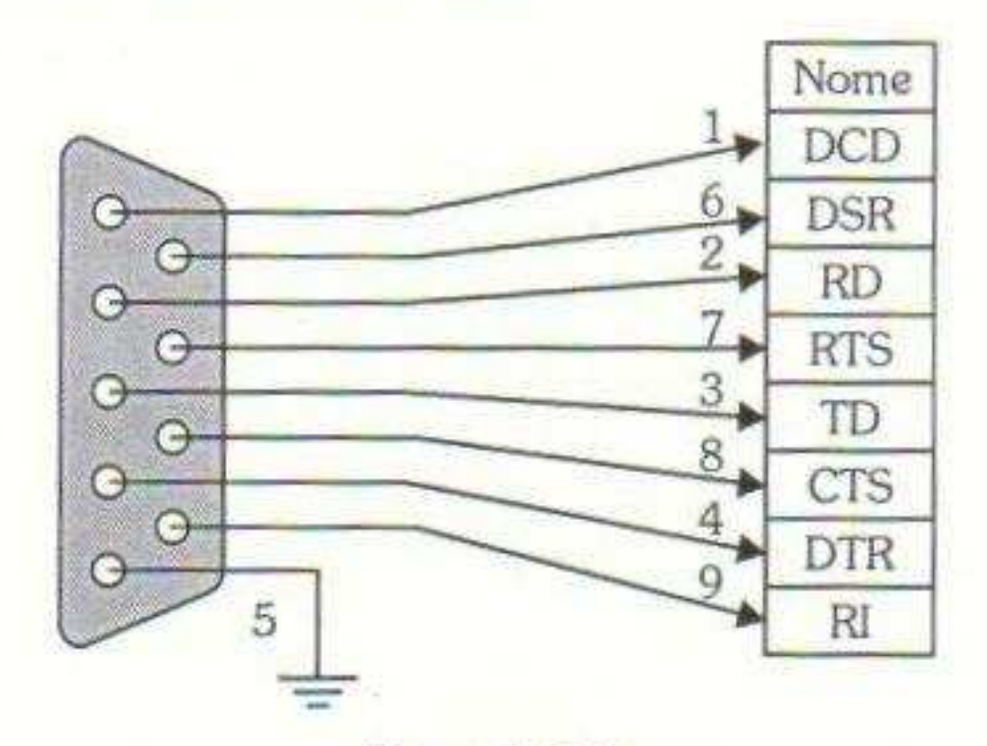

Figura 12.24

Nota: Um detalhe interessante a ser observado é que a sigla RS (Recommended Standard - Padrão recomendado), comumente utilizada como prefixo do número do protocolo (o famoso RS-232), não é mais utilizada, pois as últimas padronizações especificam que a sigla RS deve ser utilizada para abreviar a expressão Radio Sector (setor de rádio).

12.5.7.2. Interface TIA/EIA-485

O TIA/EIA-485, ou simplesmente 485, é um protocolo elétrico utilizado para comunicação de dados de alta velocidade, em distâncias de até 1200 metros, sendo adequado para ambientes expostos a elevados níveis de ruídos, como, por exemplo, as indústrias.

Trata-se de um padrão que utiliza linhas diferenciais para permitir uma maior imunidade a ruídos e também uma maior velocidade de comunicação.

O protocolo estabelece que a impedância mínima de saída do transmissor seja de 12 kohms, ou 1 UL (Unit Load - Unidade de carga) e a impedância mínima da linha seja de 375 Ohms, o que limita a um máximo de 32 dispositivos conectados simultaneamente ao barramento 485.

No entanto, no caso de utilização de transceptores com impedância de entrada superior a 12 kohms, podemos aumentar esta quantidade máxima de dispositivos no barramento. Assim, se utilizarmos transceptores com impedância de entrada na faixa de 24 kohms, podemos conectar até 64 dispositivos ao barramento, e assim por diante.

Observe que apesar de todos os dispositivos poderem atuar como transmissores ou receptores, somente é permitido o funcionamento de um transmissor de cada vez. Este controle deve ser providenciado pelo protocolo de comunicação utilizado sobre a linha 485.

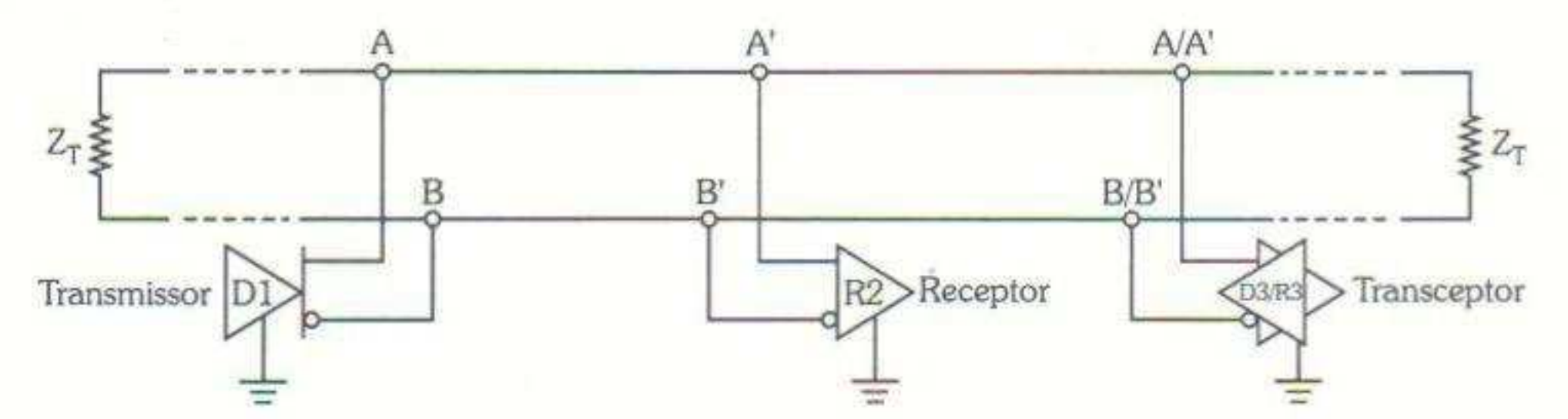

Figura 12.25

Vejamos algumas características especificadas pelo protocolo:

- Níveis de tensão: o padrão 485 estabelece uma tensão diferencial de saída mínima de 1,5 e máxima de 5 Volts;

- Comprimento do cabo: até 1200 metros (dependendo do nível de ruído);
- Número máximo de dispositivos conectados ao barramento (1UL): 32;
- Impedância de entrada mínima do receptor: 12 kohms;
- Impedância mínima de saída do transmissor: 60 Ohms;
- Velocidade de comunicação: até 10 Mbps (dependendo do nível de ruído e comprimento da linha).

12.6. Leitura de Teclas e de Teclados

De maneira geral, as teclas são a principal forma de interação humana com uma máquina ou equipamento, e sendo assim, é importante que vejamos técnicas de leitura e interface com teclados.

Uma análise simplificada dos problemas envolvidos na leitura de teclado leva à conclusão de que basta observar o nível lógico associado a uma tecla: caso ela esteja pressionada, teremos um determinado nível; caso esteja solta, teremos outro nível.

Os problemas vão além desta simples visão. O fato é que ao pressionarmos uma tecla, são gerados diversos pulsos de ruído, que atingem a entrada do microcontrolador e podem provocar erros, fazendo o software acreditar que houve diversos pressionamentos de tecla.

Para impedir este tipo de efeito, utiliza-se uma técnica de filtragem simples conhecida em inglês como "debounce". Essa filtragem consiste em detectar o pressionamento da tecla, mas somente disparar o evento relacionado à tecla, caso ela permaneça pressionada por um determinado período de tempo.

Existem diversas formas de implementar o "debounce" na leitura de uma tecla.

A forma mais simples de realizar a leitura e a filtragem de uma tecla é demonstrada em seguida. Observe o circuito da figura 12.26.

Este circuito utiliza um PIC 12F675, de 8 pinos, possuindo 1024 words de memória de programa (14 bits), 64 bytes de SRAM e 128 bytes de EEPROM, além de oscilador interno de 4MHz, um comparador analógico, conversor A/D de 4 canais e 10 bits e dois timers (um de 8 e outro de 16 bits).

Uma característica interessante desse chip é que todos os pinos de E/S (referenciados pelo registrador GPIO) podem ser configurados individualmente para ativar/desativar tanto os pull-ups internos como a interrupção por mudança de estado do pino. Para isso ele conta com dois registradores SFR específicos: WPU e IOCB.

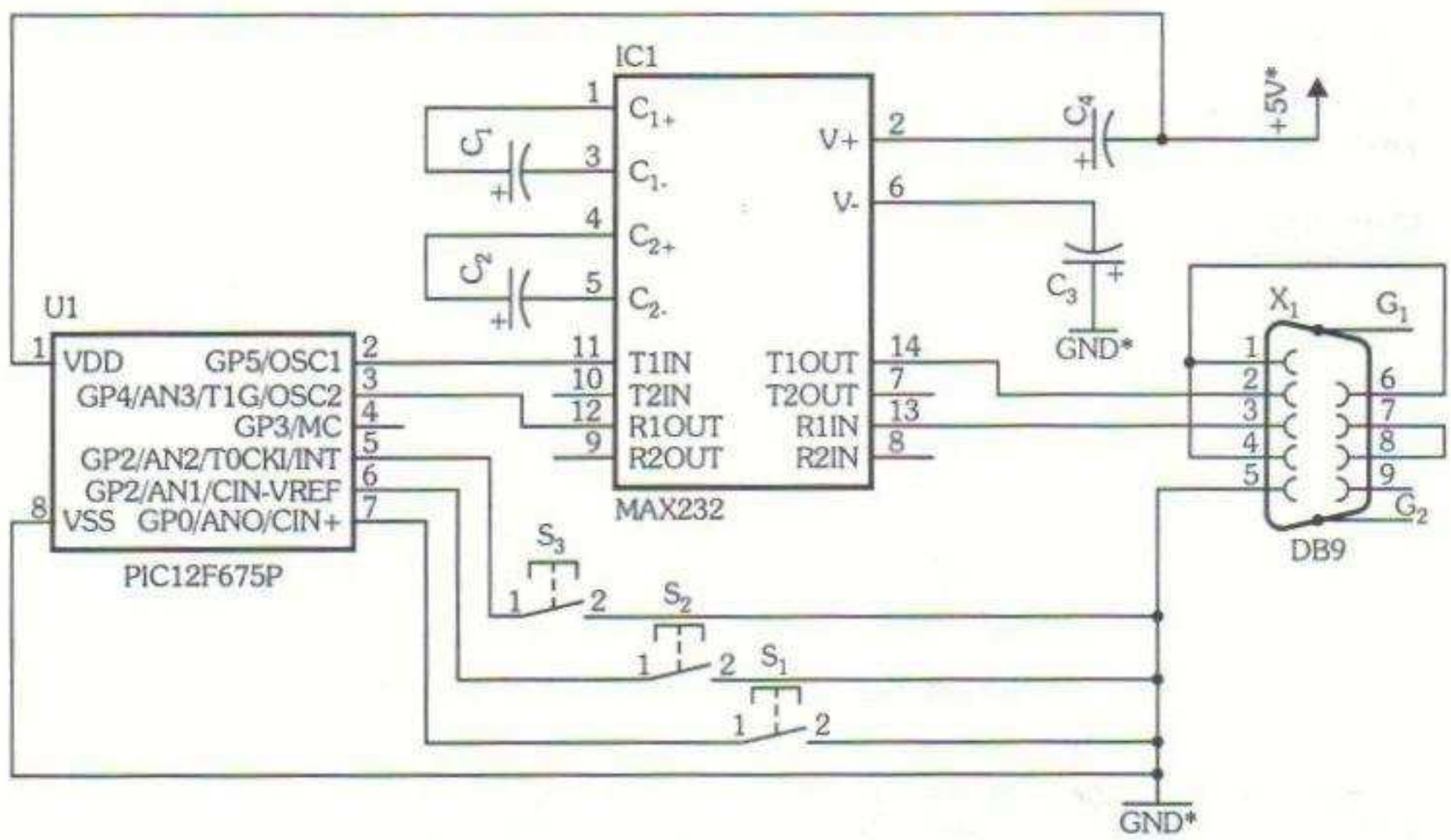

Figura 12.26

O registrador WPU (Weak Pull-Up register) é utilizado para controlar o "resistor" de pull-up interno de cada pino individualmente (desde que o pino esteja configurado como entrada). Observe que o pino GP3 não possui esta função, o que resulta na não-implementação do bit 3 do registrador.

O registrador IOCB (Interrupt On Change GPIO register) é utilizado para selecionar os pinos de E/S (configurados como entradas) que farão parte da função de interrupção por mudança de estado (GPIF).

Vejamos então o programa para verificar as teclas e enviar um código pela interface serial:

Exemplo 12.19

```
// Este programa transmite serialmente um código para cada tecla
// pressionada. A tecla S1 envia o número "1", S2 envia o número "2" e
// S3 envia o número "3".
#include <12f675.h>
#use delay(clock=4000000)
// Configura o PIC para clock interno, pinos GP4 e GP5 para IO
// watchdog desligado, Power-up timer ativado, Detector de Brownout
// desligado e MCLR interno (GP3 é I/O)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,NOMCLR
// inclui definições dos registradores do PIC12F675
#include <regs_12f6xx.h>
// valor de calibração do dispositivo, sujeito a alteração conforme o
// lote de fabricação do chip
#rom 0x3ff = { 0x34b4 }
// configurações de comunicação serial utilizando a biblioteca de
// software descrita neste livro
#define baud_rate 19200
#include <rs232.c>
```

```
#bit pino_tx = 0x05.5
#bit pino_rx = 0x05.4
#bit dir_tx  = 0x85.5
#bit dir_rx  = 0x85.4

char tcl;
boolean tecla_pres;

char varre_teclas (void)
// verifica se alguma tecla está pressionada
{
   if (!input(pin_a0))   return('1'); // verifica a tecla S1
   if (!input(pin_a1))   return('2'); // verifica a tecla S2
   if (!input(pin_a2))   return('3'); // verifica a tecla S3
   // se nenhuma tecla está pressionada, desliga o flag
   // tecla_pres e retorna o valor 0
   tecla_pres = 0;
   return (0);
}
char teclas (void)
// Rotina principal de verificação do teclado
// Verifica se é uma nova tecla e filtra ruídos
{
   int t;
   t = varre_teclas(); // verifica se há tecla
   if ((t) && (!tecla_pres))
   // se há tecla e o flag está apagado
   {
         tecla_pres = 1; // ativa o flag
         if (t != tcl)
         // se a tecla atual é diferente da tecla
         // anterior
         {
               // filtra o ruido de contato
               delay_ms (50); // aguarda 50ms
               // lê novamente as tecla e verifica
               // se a mesma tecla ainda está pressionada
               // caso positivo, retorna a tecla
               if (varre_teclas() == t) return (t);
         }
   }
   return (0); // se não há tecla, retorna 0
}
main()
{
// primeiro, lê o valor de calibração do oscilador
// e armazena no registrador OSCCAL. As versões
// mais recentes do PCM não necessitam desta rotina.
/*
#asm
   call 0x03ff
   movwf osccal
#endasm
*/
   cmcon = 0x07; // desativa entradas do comparador analógico
   ansel = 0; // configura entradas do A/D para E/S digital
   wpu = 0x07;  // ativa os pull-ups dos pinos 0, 1 e 2
   gppu = 0; // ativa o pull-up do GPIO
   rs232_inicializa();  // inicializa pinos RS232
   rs232_transmite ("Teste de Teclas\r\n");
   tcl = 0; // limpa a tecla atual
```

```
        tecla_pres = 0; // desliga o flag de tecla pressionada
        while (true)
        {
                tcl=teclas(); // verifica se há tecla pressionada
                if (tecla_pres)
                // caso exista tecla pressionada
                {
                        // verifica se é a tecla é maior que 0
                        if (tcl) rs232_transmite (tcl); // envia a tecla
                }
        }
}
```

O método descrito anteriormente funciona bem para uma pequena quantidade de teclas, mas à medida que aumentamos o número de teclas, também aumenta a quantidade de pinos de E/S necessários para a conexão do teclado.

De forma a economizar pinos de E/S, podemos agrupar as teclas em uma estrutura de matriz, que é lida por meio de um sistema de varredura, de forma a verificar cada coluna separadamente, uma após a outra.

No nosso exemplo, utilizaremos uma matriz de três colunas por quatro linhas, totalizando um máximo de 12 teclas.

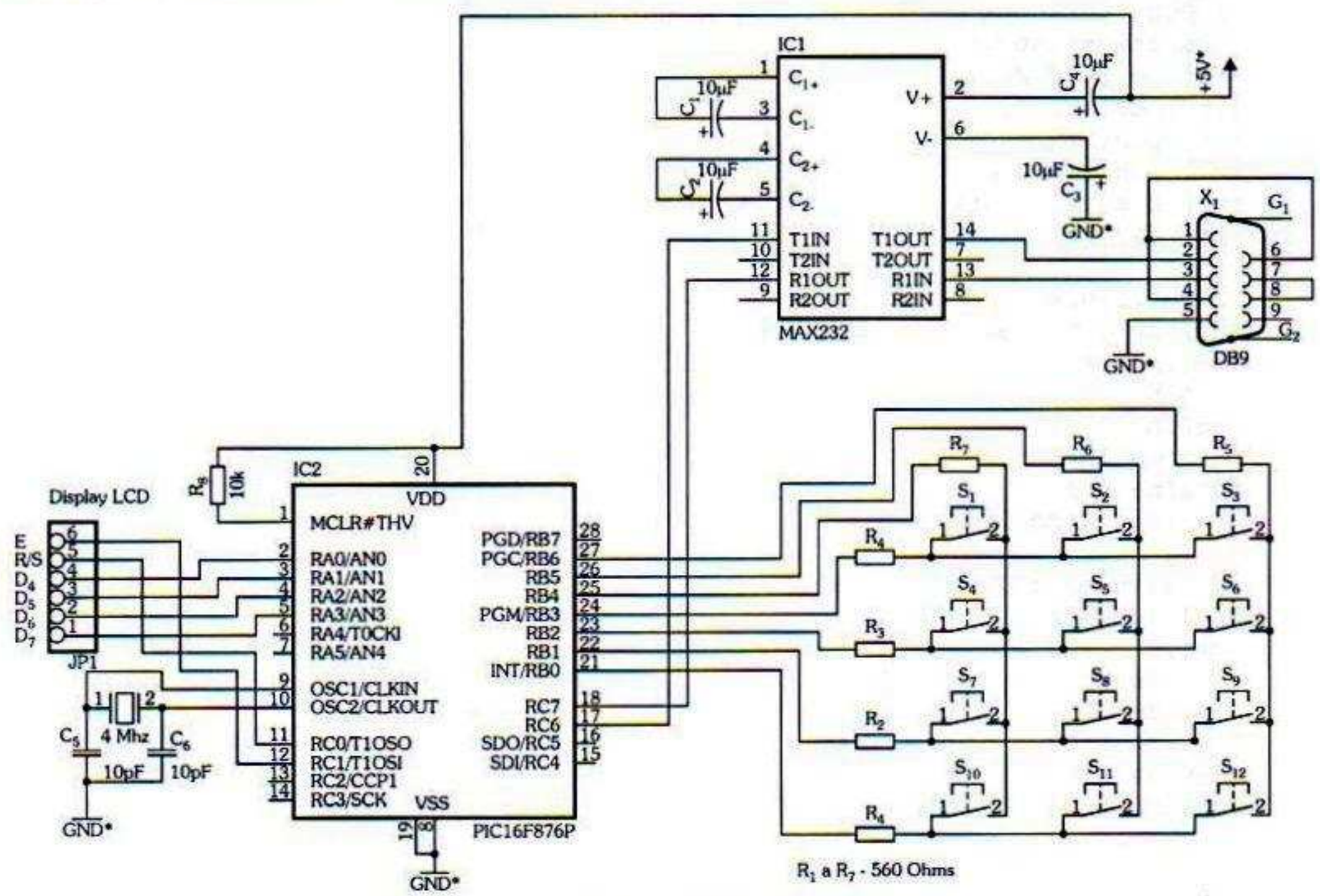

Figura 12.27

O circuito é baseado no PIC 16F876 ou 16F876A, podendo ser facilmente convertido no PIC 18F252 (que é compatível pino a pino com o 876), ou outros modelos com algumas modificações no hardware e no software.

Novamente a idéia básica é transmitir o código da tecla pressionada para o PC, por meio da linha serial, além disso, nesse projeto, a tecla pressionada será também apresentada no display LCD.

O teclado inclui ainda outras facilidades como uma tecla de segunda função, que quando pressionada em conjunto com outra tecla, atribui outro código a ela, permitindo que tenhamos um total de 22 códigos de tecla possíveis, além da auto-repetição, que repete automaticamente o caractere, caso a tecla permaneça pressionada por um determinado período de tempo.

Exemplo 12.20

```
// Este programa transmite serialmente o caractere pressionado no teclado
// além de apresentá-lo no display LCD. O teclado inclui facilidades de
// auto-repetição e função shift
#include <16f876.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT
#include <regs_16F87X.h>

// Configurações de comunicação serial utilizando a biblioteca de
// software descrita neste livro
#define baud_rate 19200
#include <rs232.c>
#bit pino_tx = 0x07.6
#bit pino_rx = 0x07.7
#bit dir_tx  = 0x87.6
#bit dir_rx  = 0x87.7

// configuração dos pinos do LCD
#define lcd_enable      pin_c1          // pino enable do LCD
#define lcd_rs          pin_c0          // pino rs do LCD
#define lcd_d4          pin_a0          // pino de dados d4 do LCD
#define lcd_d5          pin_a1          // pino de dados d5 do LCD
#define lcd_d6          pin_a2          // pino de dados d6 do LCD
#define lcd_d7          pin_a3          // pino de dados d7 do LCD
#include <mod_lcd.c>

// Definições da matriz do teclado
#define COL1     pin_b4
#define COL2     pin_b5
#define COL3     pin_b6
#define LIN1     pin_b3
#define LIN2     pin_b2
#define LIN3     pin_b1
#define LIN4     pin_b0

struct stecla
{
   int cod_tecla : 7;
   int nova : 1;
} tecla;
```

```
char varre_teclas (void)
// Realiza a varredura do teclado e retorna o
// valor da tecla pressionada.
// O teclado está conectado da seguinte forma:
//
//          COL1    COL2    COL3
// LIN1     1       2       3
// LIN2     4       5       6
// LIN3     7       8       9
// LIN4     S       0       E
//
// S - Shift    E - Enter
//
// Os valores decimais retornados são:
//          Sem shift       Com shift
// 0        48              80
// 1        49              81
// 2        50              82
// 3        51              83
// 4        52              84
// 5        53              85
// 6        54              86
// 7        55              87
// 8        56              88
// 9        57              89
// E        13              13
// Esta função possui caracter genérico
// e pode ser reescrita para otimizar
// casos específicos
{
   int tecla;
   boolean shift;
   shift = 0;
   output_high(COL1); // ativa a primeira coluna
   output_low(COL2);
   output_low(COL3);
   if (input(LIN1)) tecla = 49;
   if (input(LIN2)) tecla = 52;
   if (input(LIN3)) tecla = 55;
   if (input(LIN4)) shift = 1;
   output_low(COL1);
   output_high(COL2); // ativa a segunda coluna
   if (input(LIN1)) tecla = 50;
   if (input(LIN2)) tecla = 53;
   if (input(LIN3)) tecla = 56;
   if (input(LIN4)) tecla = 48;
   output_low(COL2);
   output_high(COL3); // ativa a terceira coluna
   if (input(LIN1)) tecla = 51;
   if (input(LIN2)) tecla = 54;
   if (input(LIN3)) tecla = 57;
   if (input(LIN4)) tecla = 13;
   if ((shift) && (tecla!=13)) tecla += 32;
   return (tecla);
}

struct stecla trata_teclas (void)
{
   // constantes de temporização do teclado
   // ciclos de filtragem de tecla
```

```
const int c_filtro = 10;
// ciclos antes de iniciar a auto-repetição
const int c_ini_autorep = 50;
// ciclos entre cada repetição de tecla
const int c_autorep = 20;
static boolean filtro, ini_autorep;
static int tempo_filtro, tempo_ini_autorep, tempo_autorep;
static int tecla_anterior;
int tecla_atual;
struct stecla tecla_pres;
tecla_atual = varre_teclas();
if (tecla_atual)
{
        if (tecla_atual == tecla_anterior)
        {
                if (filtro)
                {
                        tempo_filtro --;
                        if (!tempo_filtro)
                        {
                                tempo_filtro = c_filtro;
                                filtro = 0;
                                tecla_pres.cod_tecla = tecla_atual;
                                tecla_pres.nova = 1;
                                ini_autorep = 1;
                        }
                } else if (ini_autorep) // se filtro = 0
                {
                        tempo_ini_autorep --;
                        if (!tempo_ini_autorep)
                        {
                                tempo_ini_autorep = c_ini_autorep;
                                ini_autorep = 0;
                        }
                } else  // se ini_autorep = 0
                {
                        tempo_autorep --;
                        if (!tempo_autorep)
                        {
                                tecla_pres.cod_tecla = tecla_atual;
                                tecla_pres.nova = 1;
                                tempo_autorep = c_autorep;
                        } else tecla_pres.nova = 0;
                }
        } else // tecla_atual != tecla_anterior
        {
                tecla_anterior = tecla_atual;
                filtro = 1;
                ini_autorep = 0;
        }
} else
{
        tecla_pres.nova = 0;
        tecla_anterior = 0;
        filtro = 0;
        ini_autorep = 0;
        tempo_autorep = c_autorep;
        tempo_filtro = c_filtro;
        tempo_ini_autorep = c_ini_autorep;
}
return (tecla_pres);
```

```
}

main()
{
   // O registrador CMCON somente está implementado nas versões "A"
   //CMCON = 7;            // desabilita comparadores analógicos
   ADCON1 = 7;             // desabilita entradas analógicas
   lcd_ini ();             // inicializa display
   rs232_inicializa();     // inicializa pinos RS232
   while (true)
   {
          tecla = trata_teclas();
          if (tecla.nova)
          // caso exista tecla pressionada
          {
                 rs232_transmite (tecla.cod_tecla); // envia a tecla
                 // apresenta no display LCD
                 switch (tecla.cod_tecla)
                 {
                        // shift + 0 apaga o display
                        case 80 :       lcd_escreve ('\f');
                                                        break;
                        // ... outros comandos podem ser adicionados aqui
                        default :       lcd_escreve (tecla.cod_tecla);
                 }
          }
          delay_ms(10);  // tempo entre varreduras
   }
}
```

12.7. Apresentação em Display

Conforme vimos no tópico anterior, os teclados são uma forma muito utilizada de interação do ser humano com a máquina, no entanto eles são unidirecionais e somente permitem que o fluxo de informações flua no sentido homem → máquina, mas não o contrário.

Para que a informação seja dirigida da máquina para o homem, é necessário outra forma de interação, como os displays, por exemplo.

Neste tópico vamos abordar dois tipos diferentes de display: os displays LED de sete segmentos e os módulos LCD.

Apesar de ambos destinarem-se basicamente à mesma função, existem diversas diferenças que tornam cada um mais adequado a certo tipo de aplicação.

Os displays LED (sejam eles do tipo sete segmentos ou matriz de pontos) são mais adequados nas aplicações em que se exigem alta visibilidade, visualização de grande porte, entre outras. A desvantagem deste tipo de display é o seu elevado consumo (cada segmento de um display consome em média 10mA).

Já os displays LCD permitem uma grande concentração de informações em um pequeno espaço físico. Além disso, apresentam um baixo consumo de energia (da ordem de 1mA, quando em plena atividade e sem iluminação de fundo). Os inconvenientes deste tipo de display são a sua pequena visibilidade (displays com tamanho físico pequeno), ângulo de observação limitado, dependência de iluminação externa (natural, artificial ou de fundo) e relativamente alto custo (quando comparado a displays LED de sete segmentos).

12.7.1. Displays LED de Sete Segmentos

Basicamente, um display de sete segmentos é composto de oito LEDs, sendo sete para formar os segmentos de apresentação, mais um para o ponto decimal, e classificam-se em anodo ou catodo comum, dependendo da topologia de conexão interna dos LEDs.

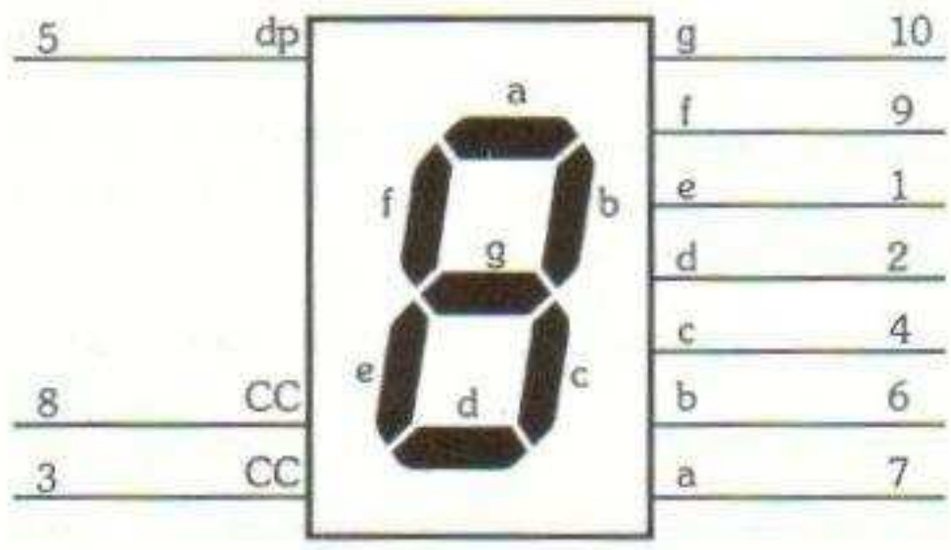

Figura 12.28

Existem diversas formas de utilizar e conectar o display. Por ora, vamos demonstrar a decodificação do display por software.

A forma mais simples e prática de implementar a decodificação do display é por meio de uma tabela.

Vejamos um exemplo simples de decodificação de display:

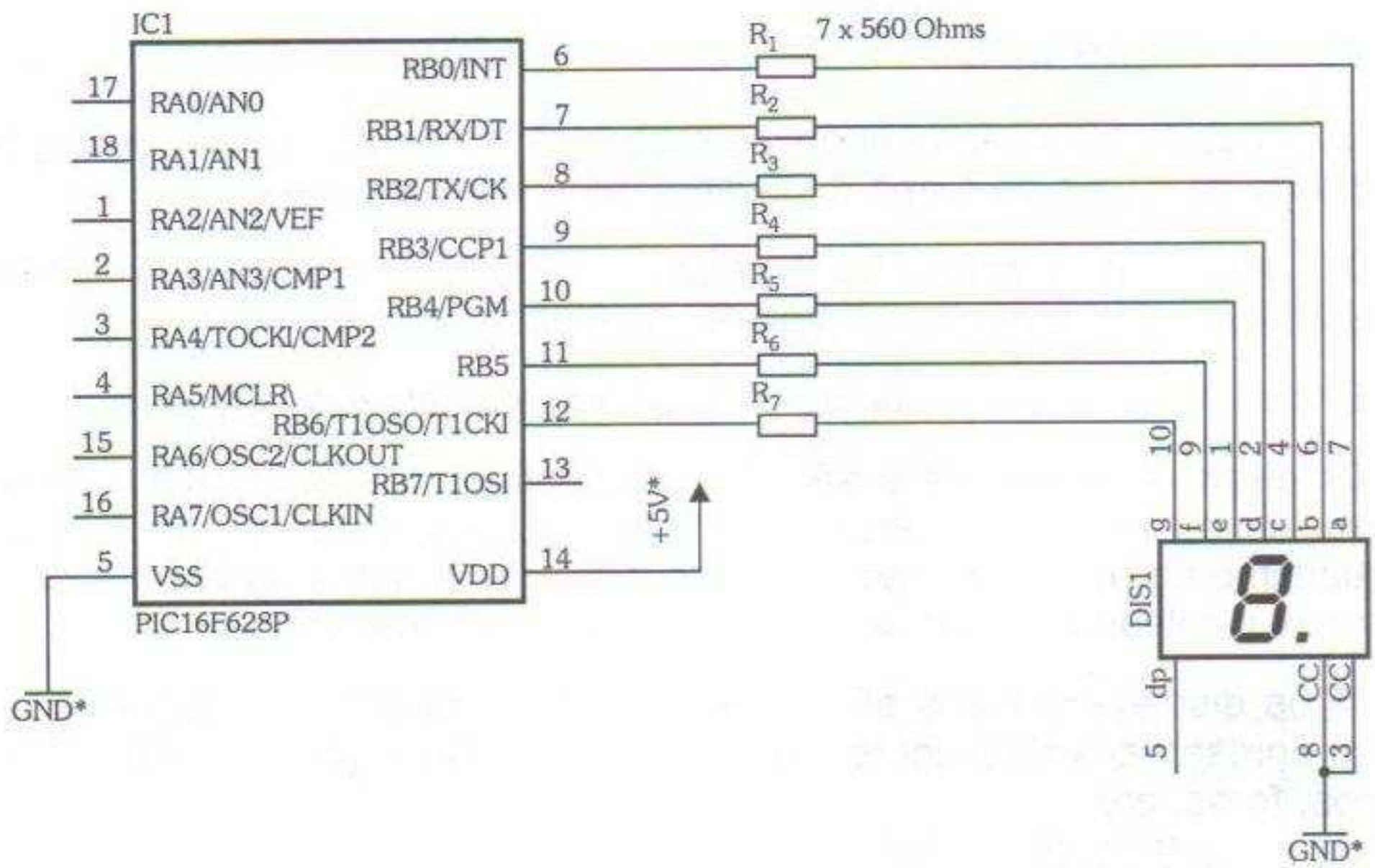

Figura 12.29

Exemplo 12.21

```
#include <16f628.h>
#use delay(clock=4000000)
#fuses INTRC_IO,NOWDT,PUT,NOBROWNOUT,NOMCLR
#include <regs_16.h>

byte const tabela[]={    0b10111111,  // 0 - 0
                         0b10000110,  // 1 - 1
                         0b11011011,  // 2 - 2
                         0b11001111,  // 3 - 3
                         0b11100110,  // 4 - 4
                         0b11101101,  // 5 - 5
                         0b11111101,  // 6 - 6
                         0b10000111,  // 7 - 7
                         0b11111111,  // 8 - 8
                         0b11100111}; // 9 - 9
main()
{
   cmcon = 7;  // configura entradas dos comparadores para digitais
   int valor = 0;
   while (true)
   {
         valor ++;
         if (valor>9) valor = 0;
         output_b (tabela[valor]); // apresenta o valor
   }
}
```

12.7.2. Módulo LCD

Os módulos LCD são uma opção muito prática de apresentar uma grande quantidade de dados de forma relativamente simples e barata.

Um módulo de display LCD é constituído basicamente de um display de cristal líquido (LCD) e de um controlador de display.

Existem dois tipos de módulo LCD: os de caractere e os gráficos.

Os displays de caractere são mais baratos e capazes de apresentar caracteres como letras, números e símbolos. Esses displays não funcionam adequadamente para a apresentação de gráficos, já que a sua tela é dividida em linhas e colunas, e cada posição armazena um caractere.

Já os displays gráficos são mais caros e complexos de programar, mas podem apresentar virtualmente qualquer tipo de informação na tela, inclusive gráficos, fotos, etc.

Neste tópico estudaremos os módulos de caractere baseados no chip controlador HD 44780, um chip que é praticamente um padrão de fato, no segmento de módulos de display LCD.

Esses controladores permitem uma interface simples com sistemas microprocessados ou microcontrolados, com largura de barramento de dados selecionável para 4 ou 8 bits, requerendo ainda mais três linhas de sinalização adicionais: ENABLE, RS e R/W.

A comunicação no modo de 4 bits é realizada utilizando apenas as quatro linhas mais significativas de dados (D7 a D4), dividindo o byte em dois nibbles que são transferidos sempre iniciando pelo mais significativo seguido pelo menos significativo.

Esses controladores dispõem ainda de 80 bytes de memória RAM (chamada de DDRAM - Data Display RAM - RAM de dados do display), 64 bytes de RAM para o gerador de caracteres do usuário (chamada de CGRAM - Caracter Generator RAM - RAM do gerador de caracteres) e 9920 bits de memória ROM do Gerador Caracteres (CGROM), perfazendo um total de 208 caracteres 5x8 mais 32 caracteres 5x10.

Observe que a CGRAM permite que o usuário construa até 8 caracteres 5x8 ou 4 caracteres 5x10. Esses caracteres podem ser utilizados para criar símbolos especiais ou outros caracteres que se desejar.

A programação de um módulo LCD de caracteres utilizando o HD 44780 (ou equivalente) obedece ao conjunto de comandos descritos na tabela 12.11:

Comando	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Descrição	Duração
Apaga Display	0	0	0	0	0	0	0	0	0	1	Apaga o display e retorna o cursor para a primeira posição da primeira linha.	-
Retorno	0	0	0	0	0	0	0	0	1	x	Retorna o cursor para a primeira posição da primeira linha e retorna a mensagem ao formato original caso estivesse deslocada.	1,52ms
Configura modo	0	0	0	0	0	0	0	1	I/D	S	I/D - Configura o sentido de deslocamento do display: 1 - movimenta o cursor à direita para cada caractere escrito, 0 - movimenta o cursor à esquerda para cada caractere escrito. S - ativa (1) ou desativa (0) o deslocamento da mensagem a cada leitura da DDRAM.	37µs
Controle Liga/Desliga do display	0	0	0	0	0	0	1	D	C	B	Liga (D=1) ou desliga (D=0) o display, ativa (C=1) ou desativa (C=0) o cursor e ativa (B=1) ou desativa (B=0) o cursor piscante.	37µs
Deslocamento	0	0	0	0	0	1	S/C	R/L	x	x	S/C - seleciona entre deslocamento do cursor (0) e cursor + mensagem (1) R/L - Sentido do deslocamento: direita (1) ou esquerda (0).	37µs
Configuração do display	0	0	0	0	1	DL	N	F	x	x	DL - número de bits do barramento: 1 - 8 bits ou 0 - 4 bits. N - número de linhas: 1 - 2 linhas, 0 - 1 linha. F - tamanho dos caracteres: 1 - 10x5, 0 - 8x5.	37µs
Endereça a CGRAM	0	0	0	1	A	A	A	A	A	A	Especifica um endereço (6 bits) de acesso à CGRAM.	37µs
Endereça a DDRAM	0	0	1	A	A	A	A	A	A	A	Especifica um endereço (7 bits) de acesso à DDRAM.	37µs
Lê contador de endereços e Busy Flag	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Lê o valor do contador de endereços (AC) e estado do flag BF: BF = 1, controlador ocupado, BF = 0, controlador livre.	37µs
Escreve dado na CGRAM ou DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Escreve um dado (D) na posição indicada pelo registrador AC da memória CGRAM ou DDRAM.	37µs
Lê um dado da CGRAM ou DDRAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Lê um dado (D) da posição de memória (CGRAM ou DDRAM) endereçada pelo registrador AC.	37µs

Tabela 12.11

Nas tabelas apresentadas em seguida, podemos observar o conjunto de caracteres disponíveis nas duas versões do controlador HD 44780. Observe que os primeiros 16 caracteres (0x00 a 0x0F) correspondem aos caracteres definidos pelo usuário na CGRAM.

Lower 4 Bits \ Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	ρ
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	ß	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	q	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	÷
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	×	₹
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	£
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	■

Tabela 12.12 - Conjunto de caracteres para ROM A00.

Lower 4 Bits \ Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)															
xxxx0001	(2)															
xxxx0010	(3)															
xxxx0011	(4)															
xxxx0100	(5)															
xxxx0101	(6)															
xxxx0110	(7)															
xxxx0111	(8)															
xxxx1000	(1)															
xxxx1001	(2)															
xxxx1010	(3)															
xxxx1011	(4)															
xxxx1100	(5)															
xxxx1101	(6)															
xxxx1110	(7)															
xxxx1111	(8)															

Tabela 12.13 - Conjunto de caracteres para ROM A02.

Para programação de caracteres na CGRAM, utiliza-se um procedimento muito simples:

Primeiramente o registrador AC é carregado com o endereço inicial da CGRAM (comando 0x40), em seguida escreve-se o dado correspondente à primeira linha do caractere a ser programado.

Lembre-se que um bit 1 significa pixel ligado e bit 0, pixel desligado. E que os três bits mais significativos (7,6 e 5) devem ser deixados em zero.

Enviam-se os dados correspondentes às próximas linhas do caractere, até a oitava ou décima, dependendo do tamanho da matriz de caractere selecionada.

Lembre-se que a última linha do caractere deve ser deixada em branco (0x00) para que o hardware possa ativar o cursor nesta posição.

Na tabela 12.14, podemos observar a construção de uma seta para cima, na primeira posição da CGRAM:

RS	R/W	DADO	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0x04	0	0	0	0	0	1	0	0
0	0	0x0E	0	0	0	0	1	1	1	0
0	0	0x15	0	0	0	1	0	1	0	1
0	0	0x04	0	0	0	0	0	1	0	0
0	0	0x04	0	0	0	0	0	1	0	0
0	0	0x04	0	0	0	0	0	1	0	0
0	0	0x04	0	0	0	0	0	1	0	0
0	0	0x00	0	0	0	0	0	0	0	0

Tabela 12.14

Lembre-se de que após efetuar a programação da CGRAM, devemos alterar o registrador AC para apontar novamente para a região da memória DDRAM.

Em seguida temos uma biblioteca genérica para interface com módulos de displays LCD.

```
/*****************************************************************/
/*  MOD_LCD.C - Biblioteca de manipulação de módulo LCD          */
/*                                                               */
/*  Autor: Fábio Pereira                                         */
/*                                                               */
/*****************************************************************/

// As definições a seguir são utilizadas para acesso aos pinos do display
// caso o pino RW não seja utilizado, comente a definição lcd_rw
#ifndef lcd_enable
   #define lcd_enable          pin_e1          // pino enable do LCD
   #define lcd_rs              pin_e0          // pino rs do LCD
   //#define lcd_rw            pin_e2          // pino rw do LCD
   #define lcd_d4              pin_d4          // pino de dados d4 do LCD
   #define lcd_d5              pin_d5          // pino de dados d5 do LCD
   #define lcd_d6              pin_d6          // pino de dados d6 do LCD
   #define lcd_d7              pin_d7          // pino de dados d7 do LCD
```

```
#endif

#define lcd_type 2            // 0=5x7, 1=5x10, 2=2 linhas
#define lcd_seg_lin 0x40     // Endereço da segunda linha na RAM do LCD

// a constante abaixo define a seqüência de inicialização do módulo LCD
byte CONST INI_LCD[4] = {0x20 | (lcd_type << 2), 0xf, 1, 6};

byte lcd_le_byte()
// lê um byte do LCD (somente com pino RW)
{
   byte dado;
   // configura os pinos de dados como entradas
   input(lcd_d4);
   input(lcd_d5);
   input(lcd_d6);
   input(lcd_d7);
   // se o pino rw for utilizado, coloca em 1
   #ifdef lcd_rw
         output_high(lcd_rw);
   #endif
   output_high(lcd_enable); // habilita display
   dado = 0;       // zera a variável de leitura
   // lê os quatro bits mais significativos
   if (input(lcd_d7)) bit_set(dado,7);
   if (input(lcd_d6)) bit_set(dado,6);
   if (input(lcd_d5)) bit_set(dado,5);
   if (input(lcd_d4)) bit_set(dado,4);
   // dá um pulso na linha enable
   output_low(lcd_enable);
   output_high(lcd_enable);
   // lê os quatro bits menos significativos
   if (input(lcd_d7)) bit_set(dado,3);
   if (input(lcd_d6)) bit_set(dado,2);
   if (input(lcd_d5)) bit_set(dado,1);
   if (input(lcd_d4)) bit_set(dado,0);
   output_low(lcd_enable);       // desabilita o display
   return dado;   // retorna o byte lido
}

void lcd_envia_nibble( byte dado )
// envia um dado de quatro bits para o display
{
   // coloca os quatro bits nas saidas
   output_bit(lcd_d4,bit_test(dado,0));
   output_bit(lcd_d5,bit_test(dado,1));
   output_bit(lcd_d6,bit_test(dado,2));
   output_bit(lcd_d7,bit_test(dado,3));
   // dá um pulso na linha enable
   output_high(lcd_enable);
   output_low(lcd_enable);
}


void lcd_envia_byte( boolean endereco, byte dado )
{
   // coloca a linha rs em 0
   output_low(lcd_rs);
   // aguarda o display ficar desocupado
   //while ( bit_test(lcd_le_byte(),7) ) ;
   // configura a linha rs dependendo do modo selecionado
```

```
    output_bit(lcd_rs,endereco);
    delay_us(100); // aguarda 100 us
    // caso a linha rw esteja definida, coloca em 0
    #ifdef lcd_rw
            output_low(lcd_rw);
    #endif
    // desativa linha enable
    output_low(lcd_enable);
    // envia a primeira parte do byte
    lcd_envia_nibble(dado >> 4);
    // envia a segunda parte do byte
    lcd_envia_nibble(dado & 0x0f);
}

void lcd_ini()
// rotina de inicialização do display
{
    byte conta;
    output_low(lcd_d4);
    output_low(lcd_d5);
    output_low(lcd_d6);
    output_low(lcd_d7);
    output_low(lcd_rs);
    #ifdef lcd_rw
            output_high(lcd_rw);
    #endif
    output_low(lcd_enable);
    delay_ms(15);
    // envia uma seqüência de 3 vezes 0x03
    // e depois 0x02 para configurar o módulo
    // para modo de 4 bits
    for(conta=1;conta<=3;++conta)
    {
            lcd_envia_nibble(3);
            delay_ms(5);
    }
    lcd_envia_nibble(2);
    // envia string de inicialização do display
    for(conta=0;conta<=3;++conta) lcd_envia_byte(0,INI_LCD[conta]);
}

void lcd_pos_xy( byte x, byte y)
{
    byte endereco;
    if(y!=1)
            endereco = lcd_seg_lin;
    else
            endereco = 0;
    endereco += x-1;
    lcd_envia_byte(0,0x80|endereco);
}

void lcd_escreve( char c)
// envia caractere para o display
{
    switch (c)
    {
      case '\f'    :        lcd_envia_byte(0,1);
                            delay_ms(2);
                            break;
```

```
      case '\n'    :
      case '\r'    :        lcd_pos_xy(1,2);
                            break;
      case '\b'    :        lcd_envia_byte(0,0x10);
                            break;
      default      :        lcd_envia_byte(1,c);
                            break;
    }
}

char lcd_le( byte x, byte y)
// le caractere do display
{
   char valor;
   // seleciona a posição do caractere
   lcd_pos_xy(x,y);
   // ativa rs
   output_high(lcd_rs);
   // lê o caractere
   valor = lcd_le_byte();
   // desativa rs
   output_low(lcd_rs);
   // retorna o valor do caractere
   return valor;
}
```

Em seguida temos dois exemplos de utilização do módulo LCD: o primeiro é um terminal RS232 simples que apresenta no display os caracteres recebidos pela linha serial, o segundo demonstra a realização de uma animação simples utilizando caracteres programados pelo usuário na CGRAM.

Exemplo 12.22

```
// Este programa ecoa o caractere enviado serialmente, além de imprimi-lo
// no display
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP
#include <mod_lcd.c>
#include <usart.c>
main()
{
   char k;
   setup_adc_ports(no_analogs);
   usart_inicializa (12,1); // velocidade: 19200
   usart_transmite ("Testando LCD + USART\r\n");
   lcd_ini();
   while (true)
   {
       k=usart_recebe();
       usart_transmite(k);
       if (k)
         if(k=='*')
            lcd_escreve('\f');
         else
            lcd_escreve(k);
   }
}
```

Exemplo 12.23

```
// Uma animação simples no display
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOBROWNOUT,NOLVP
#include <mod_lcd.c>
main()
{
   char k;
   // desliga as entradas analógicas
   setup_adc_ports(no_analogs);
   lcd_ini();
   // configura registrador AC para o endereço inicial da CGRAM
   lcd_send_byte(0,0x40);
   lcd_send_byte(1,0x04);      // primeira linha do primeiro caractere
   lcd_send_byte(1,0x0e);      // segunda linha do primeiro caractere
   lcd_send_byte(1,0x15);      // terceira linha do primeiro caractere
   lcd_send_byte(1,0x04);      // quarta linha do primeiro caractere
   lcd_send_byte(1,0x04);      // quinta linha do primeiro caractere
   lcd_send_byte(1,0x04);      // sexta linha do primeiro caractere
   lcd_send_byte(1,0x04);      // sétima linha do primeiro caractere
   lcd_send_byte(1,0x00);      // oitava linha do primeiro caractere
   lcd_send_byte(1,0x00);      // primeira linha do segundo caractere
   lcd_send_byte(1,0x04);      // segunda linha do segundo caractere
   lcd_send_byte(1,0x0e);      // terceira linha do segundo caractere
   lcd_send_byte(1,0x15);      // quarta linha do segundo caractere
   lcd_send_byte(1,0x04);      // quinta linha do segundo caractere
   lcd_send_byte(1,0x04);      // sexta linha do segundo caractere
   lcd_send_byte(1,0x04);      // sétima linha do segundo caractere
   lcd_send_byte(1,0x00);      // oitava linha do segundo caractere
   // registrador AC aponta para a primeira coluna da segunda linha
   lcd_send_byte(0,0xC0);
   while (true)
   {
      // registrador AC aponta para a primeira coluna da segunda linha
      lcd_send_byte(0,0xC0);
      // imprime o primeiro caractere do usuário
      lcd_send_byte(1,0x00);
      delay_ms(300);
      // registrador AC aponta para a primeira coluna da segunda linha
      lcd_send_byte(0,0xC0);
      // imprime o segundo caractere do usuário
      lcd_send_byte(1,0x01);
      delay_ms(300);
   }
}
```

12.8. Leitura de Tensões Analógicas com o PIC

Em muitas aplicações, pode ser necessário realizar a leitura de grandezas analógicas do mundo exterior.

Normalmente, utilizam-se sensores especiais para converter a grandeza desejada em um nível de tensão proporcional.

Podemos efetuar a leitura dessa tensão por intermédio de um conversor analógico / digital, cuja função é converter a tensão analógica em um número binário, proporcional à tensão analógica.

Nos conversores A/D, utiliza-se uma tensão de referência (normalmente chamada de Vref) que serve de fundo de escala para o sinal de saída do conversor, ou seja, as saídas do conversor estarão todas em nível "1" quando a tensão de entrada do conversor for igual ou maior que a tensão Vref.

Como a saída do conversor é um número binário proporcional à tensão medida e não o próprio valor binário da tensão, é necessário utilizar técnicas conhecidas como escalonamento, de forma a calcular o valor equivalente à saída do conversor.

O ideal seria utilizar valores de Vref que facilitem a tarefa de escalonamento, os mais comums são 1,024, 2,048 e 4,096 Volts. No entanto, como a obtenção de tais valores de tensão é complicada e muitas vezes cara (apesar de existirem muitos CIs de referência precisa de tensão), normalmente opta-se pela utilização da própria tensão de alimentação como referência para o conversor.

Para entender o funcionamento do escalonamento, suponha um conversor A/D com resolução de 10 bits e referência de 5V. Podemos facilmente concluir que o valor de cada bit será igual a $5 / (2^{10} - 1) = 4{,}8876 \ldots$ mV, ou seja, para um resultado igual a 100 (decimal), teremos uma tensão de 100 * 4,8876 mV = 0,48876 V.

Podemos realizar o escalonamento de duas formas: utilizando cálculo inteiro e utilizando cálculo de ponto flutuante.

A primeira forma é menos precisa e mais rápida.

A segunda forma é mais precisa e mais lenta (já que os cálculos em ponto flutuante ocupam muito tempo do processador).

12.8.1. Conversor A/D Interno

Como existem diversos PICs dotados de conversores A/D internos, neste tópico vamos abordar esta implementação de conversor.

Os conversores A/D padrão dos PICs são implementados utilizando a técnica de aproximação sucessiva, com resolução máxima de 10 bits, clock selecionável pelo usuário e múltiplas entradas multiplexadas.

No exemplo seguinte, utilizamos um PIC 16F877 (podendo ser utilizado o 16F877A ou ainda os 18F4x2 ou 18F4x8, mediante pequenas modificações).

O circuito foi implementado utilizando a placa de aplicações II da Symphony, tendo conectado um potenciômetro à entrada RA0. Veja o circuito básico na figura 12.30.

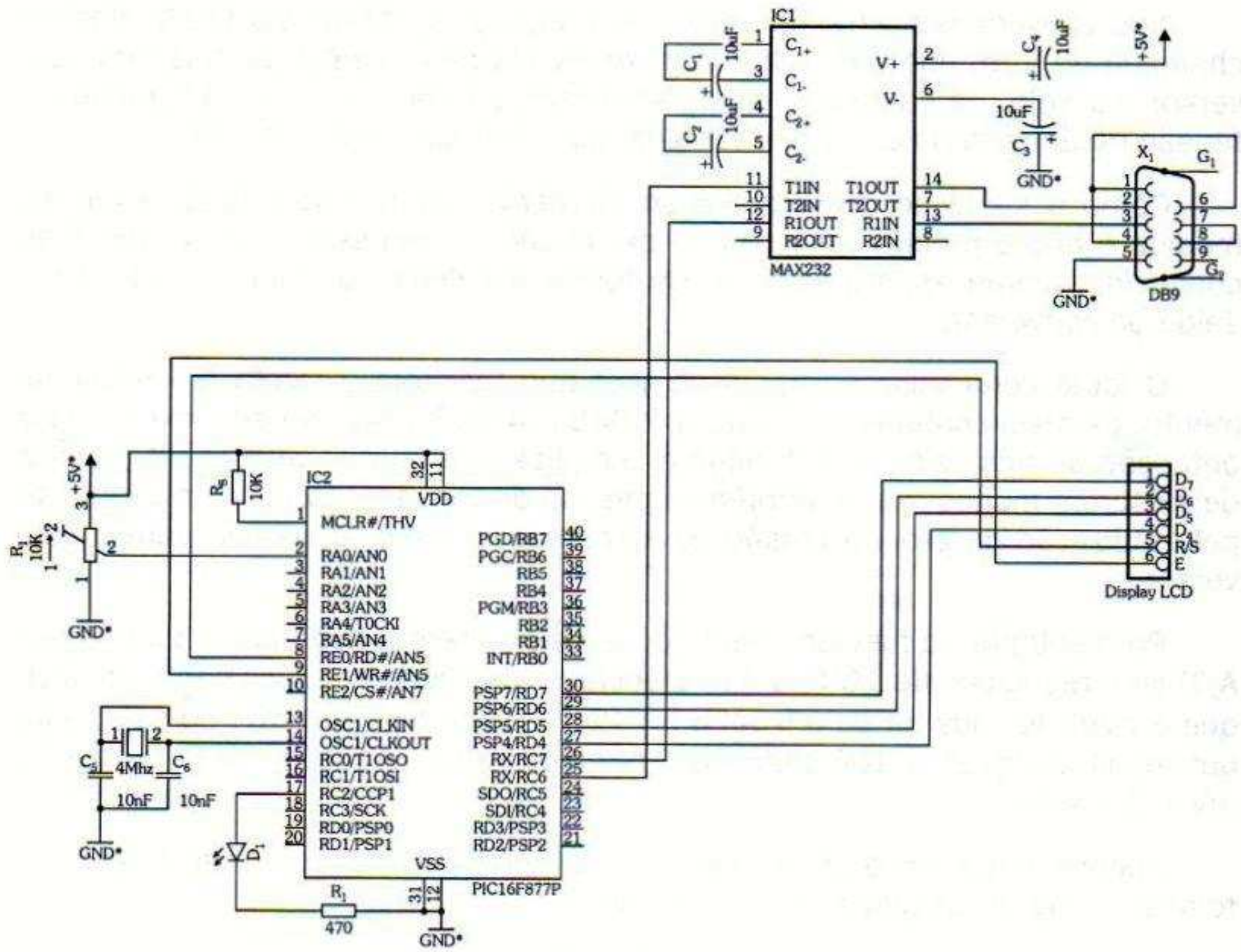

Figura 12.30

Quando trabalhamos com os conversores A/D internos dos PICs, devemos lembrar de que a impedância máxima da fonte de sinal analógico deve ser de 10 kohms.

Além disso, devemos respeitar o tempo mínimo de aquisição do circuito S/H de entrada do conversor. Este circuito é formado por um capacitor de amostragem especificado em 120 pF, além da resistência de entrada R_{IC} (especificada como sendo menor que 1 Kohm) e a resistência da chave de amostragem R_{SS} (especificada em 7 kohms @ 5 Volts).

A fórmula para cálculo do tempo mínimo de aquisição do sinal pode ser calculado pela seguinte fórmula:

$$T_{AQ} = T_{ACOMODAÇÃO} + T_{CARGA\ CAPACITOR\ DE\ AMOSTRAGEM} + \text{COEFICIENTE DE TEMPERATURA}$$

ou simplesmente: $T_{AQ} = T_{ACOM} + T_{CARGA} + CTEMP$

O tempo de carga (T_{CARGA}) do capacitor de amostragem pode ser calculado pela fórmula:

$T_{CARGA} = -(C_{HOLD}) * (R_{IC} + R_{SS} + R_{ENTRADA}) * \ln(1/2047)$

Considerando os valores especificados para C_{HOLD} (120pF), R_{IC} (1kOhm) e R_{SS} (7kOhm quando V_{DD} = 5Volts):

$T_{CARGA} = -(120pF) * (1k + 7k + R_{ENT}) * \ln(1/2047)$

Observe que R_{ENT} é o valor da impedância de entrada da fonte analógica da tensão a ser medida.

Assim, o tempo mínimo de aquisição será:

$T_{AQ}(s) = 2\mu + (-120p)*(8k+RENT)*(\ln(1/2047)) + ((TEMP_{AMB} - 25)*0{,}05\mu)$

Para uma fonte de sinal com impedância interna de 5kohms e temperatura ambiente de 40°C teríamos:

$T_{AQ}(s) = 2\mu + (-120p)*(8k+5k)*(\ln(1/2047)) + ((40 - 25)*0{,}05\mu)$

$T_{AQ}(s) = 2\mu + 11{,}894\mu + 0{,}75\mu = 14{,}644\mu s$

Na prática, temos que o tempo mínimo de aquisição para uma impedância de 50 Ohms seria de 10,61µs e para a impedância na entrada de 10 kohms, o tempo de aquisição seria de 19,72µs.

Note que é possível utilizarmos fontes de sinal com impedâncias maiores que 10 kohms, basta recorrer a uma das alternativas:

1) Aumentar o tempo de amostragem para permitir que o capacitor CHOLD consiga ser carregado com uma tensão o mais próxima possível da tensão da fonte de entrada;

2) Utilizar um circuito de entrada (normalmente um Buffer) para realizar o casamento entre a fonte de alta impedância e a entrada de baixa impedância do conversor A/D;

Exemplo 12.24

```
#include <16f877.h>
// Configura o compilador para conversor A/D de 10 bits
#device adc=10
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT
#use rs232(baud=19200, xmit=PIN_C6,rcv=PIN_C7)
#include <regs_16f87x.h>
#include <mod_lcd.c>

main()
{
   long int valor;
```

```
int32 val32;
lcd_ini();
setup_ADC_ports (RA0_analog);
setup_adc(ADC_CLOCK_INTERNAL );
set_adc_channel(0);
while (true)
{
        lcd_escreve ('\f');  // apaga o display
        // O escalonamento é realizado da seguinte forma:
        // resultado = (5000 * valor lido) / 1023
        // Para facilitar os cálculos, somamos um ao
        // valor lido:
        // resultado = (5000 * (valor + 1)) / 1024
        // simplificando:
        // resultado = ((valor + 1) * 4) + ((valor + 1) * 113) / 128
        // Repare que é necessário converter a segunda parte da
        // equação para 32 bits para que o compilador efetue o
        // cálculo corretamente
        valor = read_adc(); // efetua a conversão A/D
        // Se o valor é > 0, soma 1 ao valor lido
        if (valor) valor += 1;
        val32 = valor * 4 + ((int32)valor * 113)/128;
        // imprime o valor da tensão no display
        // 5000 = 5,000 Volts ou 5000 milivolts
        printf (lcd_escreve,"Tensao = %lu mV",val32);
        // se a tecla enter for pressionada
        if (kbhit()) if (getc() == 13)
        {
                // imprime os resultados na serial
                printf ("Tensao = %lu miliVolts\r\n",val32);
                printf ("Valor = %lu\r\n",valor);
        }
        delay_ms (250); // aguarda 250 ms
    }
}
```

12.8.2. Conversor A/D Delta - Sigma

Esta técnica de conversão pode ser implementada em praticamente qualquer microcontrolador, já que não necessita de um conversor A/D interno, apenas de um comparador analógico, que tanto pode ser interno como também conectado externamente.

O funcionamento deste tipo de conversor baseia-se em uma malha RC de entrada que atua como um integrador de tensão.

Esse integrador recebe em sua entrada a soma da tensão de entrada e uma tensão de saída fornecida por um modulador de 1 bit (ou seja, capaz de fornecer somente +5 ou 0V).

A saída do integrador é comparada com a tensão de referência e o resultado da comparação é reaplicado no modulador de 1 bit, além de ser aplicado

a um filtro digital, cujo papel é acumular o resultado, fornecendo em sua saída o resultado da conversão.

O circuito permanece nesse loop fechado em um número de ciclos igual ao número de bits total da resolução desejada. Isto significa que para uma resolução de 10 bits, são necessárias 1024 iterações do conversor.

Na figura 12.31 temos o diagrama básico da implementação do conversor A/D Delta Sigma utilizando PICs com pelo menos um comparador interno.

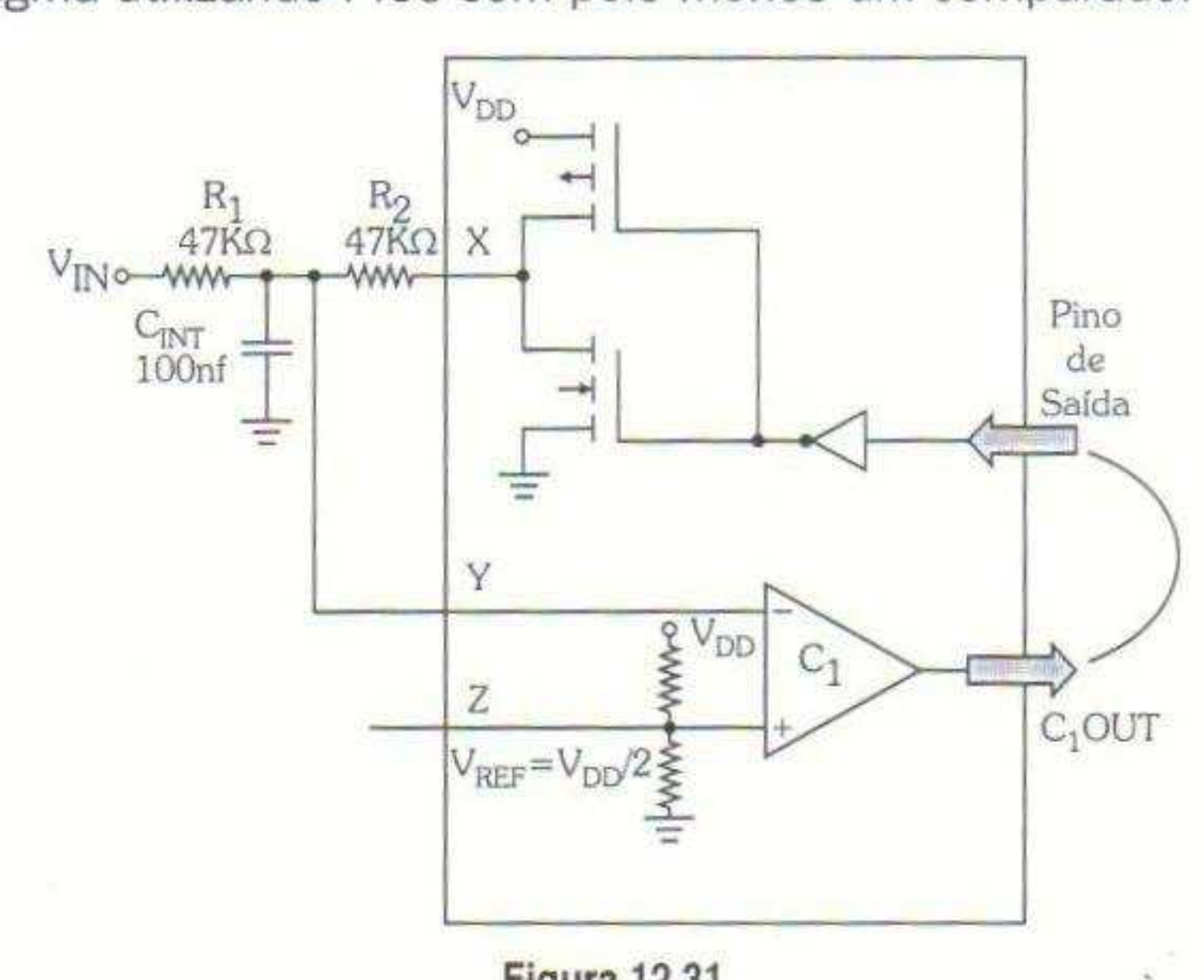

Figura 12.31

O funcionamento do circuito é muito simples: a tensão de entrada, somada à saída do modulador de 1 bit (pino de saída da figura), é aplicada ao integrador (capacitor C_{INT}).

Essa tensão de saída do integrador é comparada com a tensão de referência pelo comparador interno do PIC e o sinal de saída desse comparador é realimentado para o pino de saída "X" (modulador).

Se o modulador tiver sua saída setada, a tensão no pino "Y" irá gradualmente aumentar, até que ela seja maior que a Vref. Neste ponto, a saída do comparador vai a "0", zerando também a saída do modulador.

Isto fará com que a tensão no pino "Y" comece a diminuir até que caia abaixo de Vref, fazendo com que a saída do comparador seja setada e reiniciando o ciclo.

A cada iteração do ciclo um contador é incrementado e ao atingir $2^{resolução}$ iterações, o ciclo é interrompido.

Observe ainda que a cada iteração do ciclo em que a saída do comparador é "0", o acumulador do resultado é incrementado.

Na figura 12.32 temos o fluxograma da rotina de conversão delta-sigma. Vejamos então a sua implementação em um voltímetro simples utilizando o PIC 16F628, conforme a figura 12.33. Neste exemplo, o resultado da conversão é enviado para a interface serial cada vez que a tecla ENTER é pressionada.

Observe que a faixa de medição é de 0 a 5V, para outras faixas basta recalcular o divisor de tensão na entrada do conversor, para maiores detalhes, consulte a nota de aplicação AN700 da Microchip (o endereço eletrônico encontra-se nas referências bibliográficas do livro).

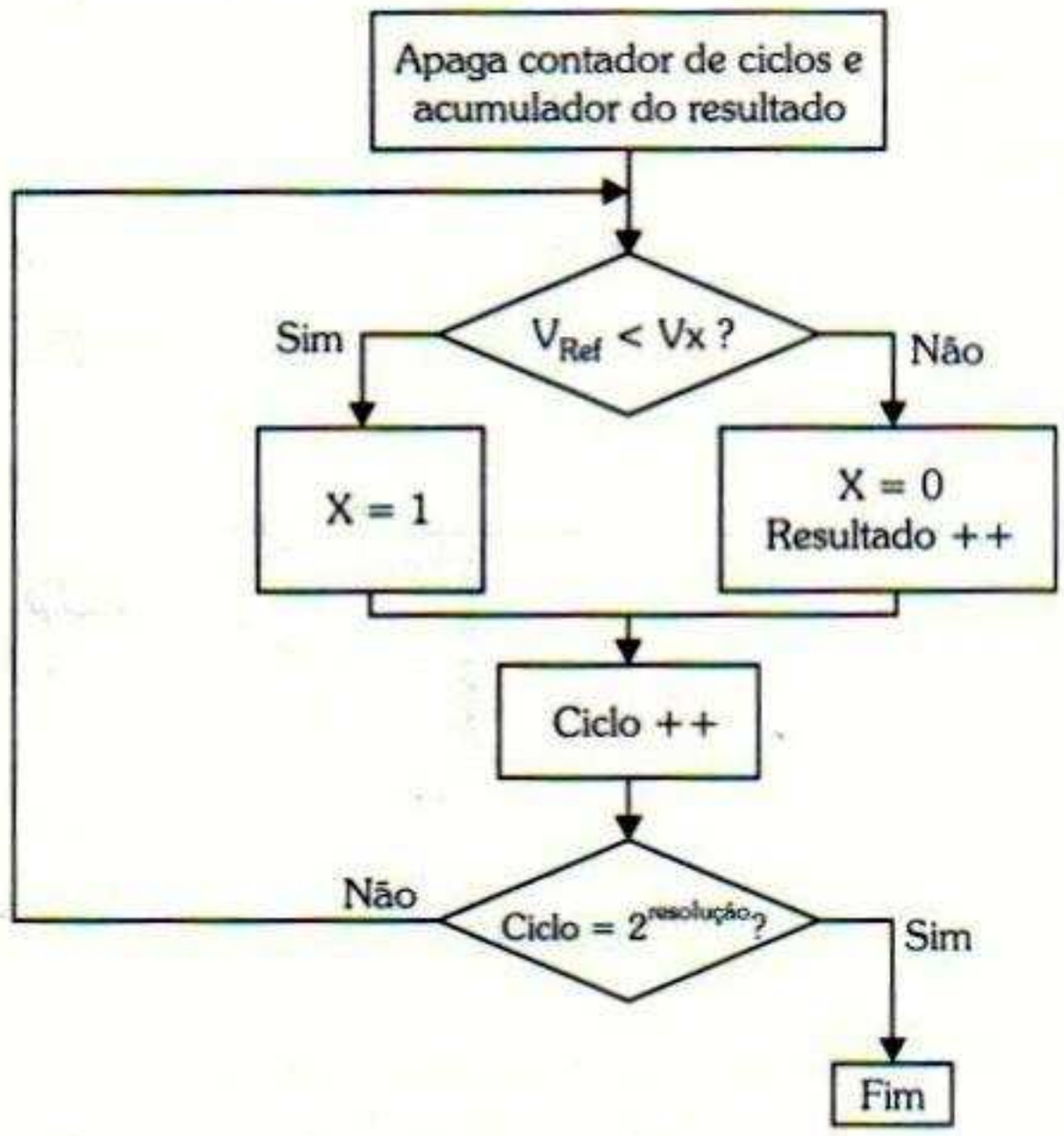

Figura 12.32

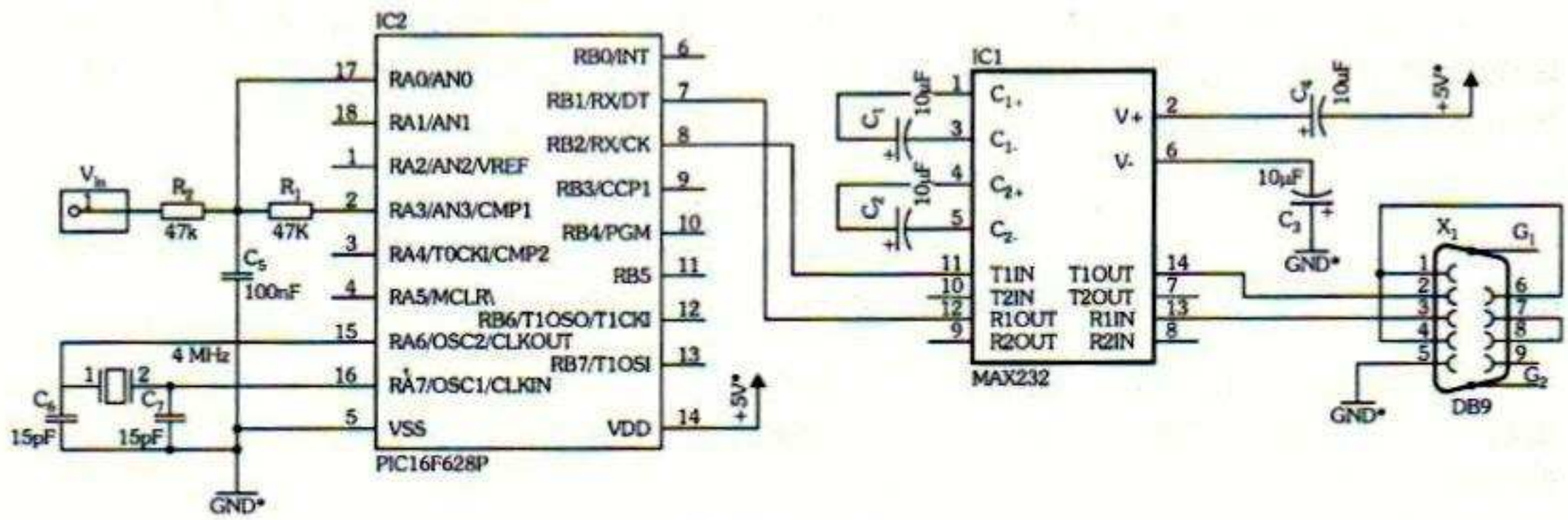

Figura 12.33

Exemplo 12.25

```
#include <16f628.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT,NOMCLR,NOBROWNOUT,NOLVP
#use rs232(baud=19200, xmit=PIN_B2,rcv=PIN_B1)
#include <regs_16.h>

#bit saida_comp = 0x1F.6 // C1OUT
#bit pino_modul = 0x05.3 // RA3
#bit tris_modul = 0x85.3 // TRISA3
#byte cmcon_reg = 0x1F
#byte vrcon_reg = 0x9F

void inicializa_delta_sigma()
{
   vrcon_reg = 0xEC;
   tris_modul = 0;
   cmcon_reg = 6;
}

long int adc_delta_sigma (long int resolucao)
// Função de conversão analógico digital utilizando a técnica delta-sigma
// Para garantir que a cada ciclo possua a mesma duração, utiliza-se o
// timer 0, atuando como um contador de instruções. No início do ciclo o
// timer 0 é iniciado com o valor 247. Ao final do loop, antes de retornar
// ao seu início, o programa aguarda que o T0IF seja setado
{
   long int resultado = 0;
   t0cs = 0;  // timer 0 com clock interno
   psa = 1;  // clock tmr0 = 1MHz
   cmcon_reg = 3;
   while (resolucao)
   {
         tmr0 = 247;  // configura o timer 0
         t0if = 0;  // apaga flag do timer 0
         if (saida_comp)
         {
               pino_modul = 1;
         } else
         {
               pino_modul = 0;
               resultado ++;  // incrementa acumulador de resultado
         }
         resolucao --; // decrementa o contador de ciclos
         while (!t0if);
   }
   cmcon_reg = 6;
   return (resultado);
}

main()
{
   long int valor;
   float val;
   printf ("\r\nADC delta sigma\r\n");
   inicializa_delta_sigma();
   while (true)
   {
         // a conversão será de 14 bits (16384)
```

```
            valor = adc_delta_sigma(16384);
            val = 5 * (float) valor / 16384;
            // se a tecla enter for pressionada
            if (kbhit()) if (getc() == 13)
            {
                    // imprime os resultados na serial
                    printf ("Tensao = %1.4f Volts\r\n",val);
                    printf ("Valor = %lu\r\n",valor);
            }
    }
}
```

12.9. Módulo PWM

Neste tópico vamos demonstrar um pequeno exemplo de um controle de brilho de LED utilizando a funcionalidade dos módulos CCP disponíveis em alguns PICs.

O exemplo utiliza uma porta serial para receber os comandos de controle que selecionam o brilho do LED conectado ao pino RC2 de um PIC 16F877.

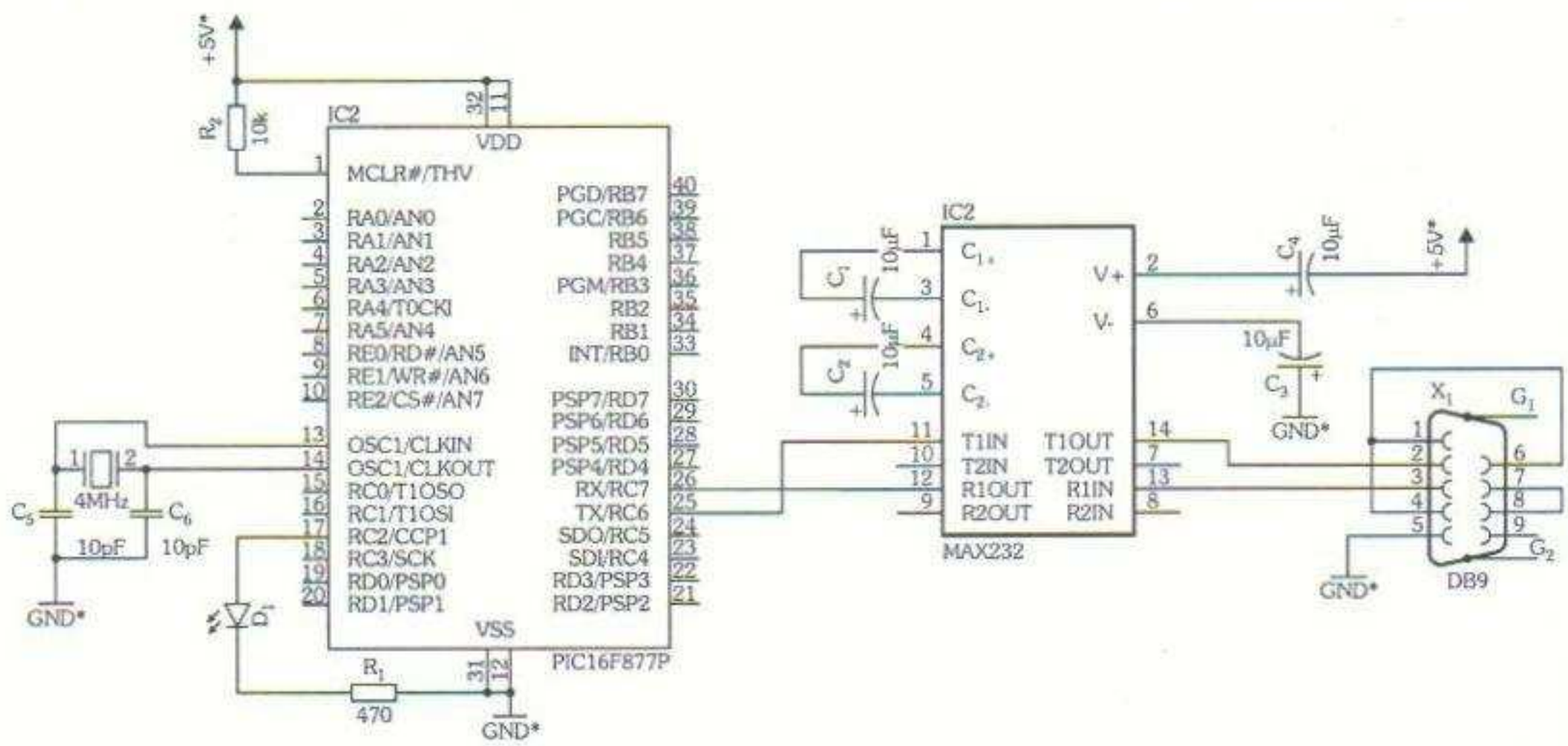

Figura 12.34

Exemplo 12.26

```
#include <16f877.h>
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT
#use rs232(baud=19200, xmit=PIN_C6,rcv=PIN_C7)
#include <regs_16f87x.h>

main()
{
```

```
long int ciclo=0;
printf ("\r\nTeste do PWM\r\n");
setup_timer_2 (T2_DIV_BY_4, 248, 1);  // timer 2 = 1,004 khz
setup_ccp1 (ccp_pwm);  // configura CCP1 para modo PWM
set_pwm1_duty ( 0 );  // configura o ciclo ativo em 0 (desligado)
while (true)
{
        if (kbhit())  // se uma tecla for pressionada
        {
                switch (getc())  // verifica a tecla
                {
                        case '1' :      ciclo = 50;
                                        break;
                        case '2' :      ciclo = 100;
                                        break;
                        case '3' :      ciclo = 255;
                                        break;
                        case '4' :      ciclo = 350;
                                        break;
                        case '5' :      ciclo = 500;
                                        break;
                        case '6' :      ciclo = 700;
                                        break;
                        case '7' :      ciclo = 900;
                                        break;
                        case '8' :      ciclo = 1023;
                                        break;
                        case '0' :      ciclo = 0;
                                        break;
                        case '-' :      ciclo -= 10;
                                        break;
                        case '=' :      ciclo += 10;
                                        break;
                }
                if (ciclo>1023) ciclo = 1023;
                printf ("Ciclo ativo = %lu\r\n",ciclo);
                set_pwm1_duty (ciclo);  // configura o ciclo ativo
                // imprime o estado do CCPR1L
                printf ("CCPR1L = %u\r\n", ccpr1l);
                // imprime o estado do bit CCP1X
                printf ("CCP1X = %u\r\n", ccp1x);
                // imprime o estado do bit CCP1Y
                printf ("CCP1Y = %u\r\n", ccp1y);
        }
    }
}
```

Observe que graças a compatibilidade pino a pino entre os modelos da série 18 e da série 16, podemos substituir o PIC 16F877 utilizado na figura 12.32 por um modelo equivalente da série 18 (18F442 ou 18F452) e utilizá-lo com um mínimo de modificação de software.

Para utilizar um 18F452, as modificações de código necessárias limitam-se a apenas duas linhas do cabeçalho do programa:

Exemplo 12.27

```
#include <18f452.h>  // seleciona o PIC 18F452
#use delay(clock=4000000)
#fuses HS,NOWDT,PUT
#use rs232(baud=19200, xmit=PIN_C6,rcv=PIN_C7)
#include <regs_18fxx2.h>  // definições de registradores da série 18

main()
{
   long int ciclo=0;
   printf ("\r\nTeste do PWM\r\n");
   setup_timer_2 (T2_DIV_BY_4, 248, 1);  // timer 2 = 1,004 khz
   setup_ccp1 (ccp_pwm);  // configura CCP1 para modo PWM
   set_pwm1_duty ( 0 );  // configura o ciclo ativo em 0 (desligado)
   while (true)
   {
         if (kbhit())  // se uma tecla for pressionada
         {
               switch (getc())  // verifica a tecla
               {
                     case '1' :     ciclo = 50;
                                    break;
                     case '2' :     ciclo = 100;
                                    break;
                     case '3' :     ciclo = 255;
                                    break;
                     case '4' :     ciclo = 350;
                                    break;
                     case '5' :     ciclo = 500;
                                    break;
                     case '6' :     ciclo = 700;
                                    break;
                     case '7' :     ciclo = 900;
                                    break;
                     case '8' :     ciclo = 1023;
                                    break;
                     case '0' :     ciclo = 0;
                                    break;
                     case '-' :     ciclo -= 10;
                                    break;
                     case '=' :     ciclo += 10;
                                    break;
               }
               if (ciclo>1023) ciclo = 1023;
               printf ("Ciclo ativo = %lu\r\n",ciclo);
               set_pwm1_duty (ciclo);  // configura o ciclo ativo
               // imprime o estado do CCPR1L
               printf ("CCPR1L = %u\r\n", ccpr1l);
               // imprime o estado do bit CCP1X
               printf ("CCP1X = %u\r\n", ccp1x);
               // imprime o estado do bit CCP1Y
               printf ("CCP1Y = %u\r\n", ccp1y);
         }
   }
}
```

Tabela ASCII

A.1. Tabela de Conversão Decimal / Hexa / ASCII

Para utilizar a tabela seguinte, selecione o caractere desejado (por exemplo, 'A'). Em seguida, verifique o número na coluna HEX que corresponde à linha em que está o caractere (linha 4). Feito isso, localize o número da linha HEX que corresponde à coluna em que está o caractere (coluna 1). Isto significa que a letra 'A' codificada em ASCII hexadecimal equivale ao número 0x41. Para decimal, basta somar o número 64 ao número 1, o que resulta em 65 decimal.

Dec		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
128	8																
144	9																
160	A																
176	B																
192	C																
208	D																
224	E																
240	F																

Anotações

Funções C - Referência Rápida

B.1. Sumário de Funções C

Nome	Retorna	Descrição
abs (int8 x)	int8	Retorna o valor absoluto de x
acos (float x)	float	Calcula o arco cosseno de x
asin (float x)	float	Calcula o arco seno de x
assert (cond)	-	Emite mensagem caso a condição (cond) seja verdadeira
atan (float x)	float	Calcula o arco tangente de x
atan2 (float x , float y)	float	Calcula o arco tangente de y / x
atof (char x[])	float	Converte a string x em um número float
atoi (char x[])	int8	Converte a string em um inteiro
atol (char x[])	int16	Converte a string em um inteiro de 16 bits
atoi32 (char x[])	int32	Converte a string em um inteiro de 32 bits
bit_clear (variável , bit)	-	Apaga o bit (bit) da variável (variável)
bit_set (variável , bit)	-	Ativa o bit (bit) da variável (variável)
bit_test (variável , bit)	int1	Retorna o valor do bit (bit) da variável (variável)
ceil (float x)	float	Retorna o menor inteiro acima de x
cos (float x)	float	Calcula o cosseno de x radianos
cosh (float x)	float	Calcula o cosseno hiperbólico de x radianos
delay_cycles (const int8 x)	-	Aguarda x ciclos de máquina
delay_ms (const long int8 x)	-	Aguarda x milissegundos
delay_us (const long int8 x)	-	Aguarda x microssegundos
disable_interrupts (inter)	-	Desliga a interrupção especificada
enable_interrupts (inter)	-	Habilita a interrupção especificada
exp (float x)	float	Calcula e elevado a x
ext_int_edge (fonte , borda)	-	Configura a borda de sensibilidade da interrupção externa
fabs (float x)	float	Retorna o valor absoluto de x
floor (float x)	float	Retorna o menor valor inteiro abaixo de x

Nome	Retorna	Descrição
fmod (float x, float y)	float	Retorna o resto inteiro da divisão float de x por y
frexp (float x, signed int8 *y)	float	Retorna o componente exponencial e fracionário de x
get_timerx ( )	int8 ou int16	Lê o conteúdo do timer x
getc ( )	char	Lê um caractere do dispositivo de entrada padrão
fgetc (stream)	char	Lê um caractere da stream (stream)
gets ( )	&char	Lê uma string do dispositivo de entrada padrão
fgets (stream)	&char	Lê uma string da stream (stream)
goto_address (int16 ou int32 x)	-	Desvia para o endereço x
i2c_poll ()	int1	Verifica se houve recepção na I2C
i2c_read (int1 ack)	int8	Lê um dado da I2C
i2c_start ( )	-	Inicia uma condição start na I2C
i2c_stop ( )	-	Insere uma condição stop na I2C
i2c_write (int8 x)	int1	Escreve um dado (x) na I2C e retorna um ack ou nack
input (int8 pino)	int1	Lê o estado de um pino do dispositivo
input_x ( )	int8	Lê o estado de uma porta do dispositivo
isamoung (char carac , const char[] x)	int1	Retorna 1 caso o caractere pertença ao conjunto x
isalnum (char carac)	int1	Testa se o argumento é uma letra ou número
isalpha (char carac)	int1	Testa se o argumento é uma letra
isdigit (char carac)	int1	Testa se o argumento é um número
islower (char carac)	int1	Testa se o argumento é uma letra minúscula
isspace (char carac)	int1	Testa se o argumento é um espaço
isupper (char carac)	int1	Testa se o argumento é uma letra maiúscula
isxdigit (int8 carac)	int1	Testa se o argumento é um dígito hexadecimal
iscntrl (char carac)	int1	Testa se o argumento é um caractere de controle
isgraph (char carac)	int1	Testa se o argumento é um caractere gráfico
isprint (char carac)	int1	Testa se o argumento é imprimível
ispunct (char carac)	int1	Testa se o argumento é um caractere de pontuação
kbhit ( )	int1	Retorna se há caractere recebido na serial
label_address (rótulo)	int16 ou int32	Retorna o endereço físico do rótulo especificado
labs (int16 x)	int16	Retorna o valor absoluto de x
ldexp (float x , signed int8 y)	float	Retorna x vezes 2 elevado a y
log (float x)	float	Calcula o logarítmo natural de x
log10 (float x)	float	Calcula o logarítmo base 10 de x
make8 (int16 ou int32 val , desloc)	int8	Lê um byte especificado por (desloc) valor (val)
make16 (int8 var1 , int8 var2)	int16	Gera um número de 16 bits a partir de dois de 8
make32 (int8 ou int16 a, b, c, d)	int32	Gera um número de 32 bits a partir de 2 ou mais de 8 ou 16 bits
memcpy (dest , fonte , num)	-	Copia n bytes do ponteiro (fonte) para o ponteiro (dest)
memset (dest , valor , n)	-	Seta n bytes do ponteiro (dest) com o valor (valor)
modf (float x, float *y)	float	Retorna a parte inteira e fracionária de x

Nome	Retorna	Descrição
offsetof (estrutura , campo)	int8	Retorna o deslocamento em bytes do campo na estrutura especificada
offsetofbit (estrutura , campo)	int8	Retorna o deslcoamento em bits do campo na estrutura especificada
output_bit (int8 pino , int1 valor)	-	Coloca o pino especificado no nível lógico (valor)
output_float (int8 pino)	-	Coloca o pino em estado de alta impedância
output_high (int8 pino)	-	Coloca o pino especificado em nível 1
output_low (int8 pino)	-	Coloca o pino especificado em nível 0
output_x (int8 valor)	-	Coloca o (valor) na porta especificada
perror (string)	-	Imprime a string de erro no dispositivo STDERR
port_b_pullups (int1 valor)	-	Configura os resistores de pull-up internos da porta b
pow (float x, float y)	float	Retorna x elevado a y
printf ([func,] string [, valores])	-	Imprime dados
fprintf (string [, valores], stream)	-	Imprime dados na stream especificada
psp_output_full ( )	int1	Testa se a porta paralela escrava já foi lida
psp_input_full ( )	int1	Testa se foi escrito um dado na porta paralela escrava
psp_overflow ( )	int1	Testa se foi sobrescrito o caractere na porta paralela escrava
putc (int8 dado)	-	Escreve o dado na saída padrão
fputc (int8 dado , stream)	-	Escreve o dado na stream especificada
puts (string)	-	Escreve a string na saída padrão
fputs (string , stream)	-	Escreve a string na stream especificada
read_adc ( )	in8 ou int16	Lê o resultado da conversão AD
read_bank (banco , desloc)	int8	Lê um registrador GPR do banco especificado
read_calibration (int8 n)	int8	Lê um valor de calibração do PIC 14000
read_eeprom (int8 end)	int8	Lê um endereço da memória EEPROM
read_program_eeprom (int16 ou int 32 end)	int8 ou int16	Lê um endereço da memória de programa
reset_cpu ( )	-	Reinicia o processador
restart_cause ( )	int8	Retorna a causa do reset do processador
restart_wdt ( )	-	Reinicia a contagem do watchdog
rotate_left (var , bytes)	-	Rotaciona 1 bit à esquerda a variável (var)
rotate_right (var , bytes)	-	Rotaciona 1 bit à direita a variável (var)
set_adc_channel (canal)	-	Seleciona um canal do AD
set_pwmx_duty (int8 ou int 16 valor)	-	Configura o ciclo ativo do pwm especificado por x
set_timerx (int8 ou int16 valor)	-	Carrega um valor no timer especificado
set_tris_x (int8 valor)	-	Configura o registrador tris especificado
set_uart_speed (valor)	-	Configura a velocidade da USART
setup_adc (int8 modo)	-	Configura o módulo ADC interno
setup_adc_ports (valor)	-	Configura os pinos de entrada analógicos
setup_ccpx (modo)	-	Configura o módulo CCP especificado
setup_comparator (modo)	-	Configura o módulo comparador analógico interno
setup_counters (modo , presc)	-	Configura o timer 0 e o prescaler

Nome	Retorna	Descrição
setup_psp (modo)	-	Configura a porta paralela escrava
setup_spi (modo)	-	Configura o SSP ou MSSP para um modo SPI
setup_timer_x (modo)	-	Configura o timer especificado
setup_vref (modo | valor)	-	Configura o módulo de referência interna de tensão
setup_wdt (modo)	-	Configura o watchdog
shift_left (vari , int8 bytes , int1 valor)	int1	Desloca a variável (vari) à esquerda inserindo o bit (valor)
shift_right (vari , int8 bytes, int1 valor)	int1	Desloca variável (vari) à direita inserindo o bit (valor)
sin (float x)	float	Retorna o seno de um ângulo de x radianos
sinh (float x)	float	
sleep ( )	-	Coloca o PIC em modo SLEEP
spi_data_is_in ( )	int1	Retorna 1 se há dado disponível na interface SPI
spi_read (int8 valor)	int8	Lê um dado da interface SPI
spi_write (int8 dado)	-	Escreve um dado na interface SPI
sprintf (string , constante , valores)	-	Imprime dados para uma string
sqrt (float x)	float	Retorna a raiz quadrada de x
strcat (s1, s2)	pointer	Concatena a string s2 em s1
strchr (s1, char c)	pointer	Retorna um ponteiro para a ocorrência do caractere c na string s1
strrchr (s1, char c)	pointer	Retorna um ponteiro para a ocorrência do caractere c na string s1, inicia a procura pelo final da string
strcmp (s1, s2)	int8	Compara a string s1 com s2
strncmp (s1, s2, int n)	int8	Compara os n primeiros caracteres de s1 com s2
stricmp (s1, s2)	int8	Compara a string s1 com s2 ignorando maiúsculas e minúsculas
strncpy (s1, s2, int n)	pointer	Copia n caracteres de s2 para s1
strcspn (s1, s2)	int8	Conta os caracteres iniciais de s1 que não pertencem a s2
strspn (s1, s2)	int8	Conta os caracteres inicias de s1 presentes em s2
strlen (s1)	int8	Conta o número de caracteres em s1
strlwr (s1)	pointer	Converte a string s1 em minúsculas
strpbrk (s1, s2)	pointer	Procura na string s1 pelo primeiro caractere também presente em s2
strstr (s1, s2)	pointer	Procura a string s2 na string s1
strtok (s1, s2)	pointer	Procura a próxima ocorrência de um sinal definido em s2 na string s1
strcpy (s1, s2)	-	Copia a string s2 na string s1
swap (int8 x)	-	Inverte os nibbles da variável x
tan (float x)	float	Calcula a tangente de x radianos
tanh (float x)	float	Calcula a tangente hiperbólica de x radianos
tolower (char x)	char	Converte o caractere x em minúsculo
toupper (char x)	char	Converte o caractere x em maiúsculo
write_bank (banco, desloc, valor)	-	Escreve um dado num endereço de memória RAM
write_eeprom (ender , valor)	-	Escreve um valor na memória EEPROM

Instruções PIC

C.1. 14 BITS

Mnemônico		Descrição	Ciclos	Código de 14 bits MSb			LSb	Flags afetados
Operações com registrador orientadas a byte								
ADDWF	f,d	Adiciona o valor de W ao valor de f (d=f+W)	1	00	0111	dfff	ffff	C,DC,Z
ANDWF	f,d	AND lógico de W com f (d=f and W)	1	00	0101	dfff	ffff	Z
CLRF	f	Limpa conteúdo de f (zera bits) (f=0)	1	00	0001	1fff	ffff	Z
CLRW		Limpa conteúdo de W (zera bits) (W=0)	1	00	0001	0xxx	xxxx	Z
COMF	f,d	Complementa f (inverte bits) (d= $\overline{f}$)	1	00	1001	dfff	ffff	Z
DECF	f,d	Decrementa f (d=f-1)	1	00	0011	dfff	ffff	Z
DECFSZ	f,d	Decrementa f (d=f-1) e pula se igual a zero	1 (2)	00	1011	dfff	ffff	Nenhum
INCF	f,d	Incrementa f (d=f+1)	1	00	1010	dfff	ffff	Z
INCFSZ	f,d	Incrementa f (d=f+1) e pula se igual a zero	1 (2)	00	1111	dfff	ffff	Nenhum
IORWF	f,d	OR lógico entre W e f (d=f or W)	1	00	0100	dfff	ffff	Z
MOVF	f,d	Copia o valor de f para o destino d (d=f)	1	00	1000	dfff	ffff	Z
MOVWF	f	Copia o valor de W para f (f=W)	1	00	0000	1fff	ffff	Nenhum
RLF	f,d	Desloca f uma posição à esquerda	1	00	1101	dfff	ffff	C
RRF	f,d	Desloca f uma posição à direita	1	00	1100	dfff	ffff	C
SUBWF	f,d	Subtrai W de F (d=f-W)	1	00	0010	dfff	ffff	C,DC,Z
SWAPF	f,d	Troca Nibbles de f (xxxxyyyy -> yyyyxxxx)	1	00	1110	dfff	ffff	Nenhum
XORWF	f,d	XOR lógico entre W e f (d=f xor W)	1	00	0110	dfff	ffff	Z
Operações com registrador orientadas a bit								
BCF	f,b	Limpa bit b do registrador f	1	01	00bb	bfff	ffff	Nenhum
BSF	f,b	Seta bit b do registrador f	1	01	01bb	bfff	ffff	Nenhum
BTFSC	f,b	Testa bit b do registrador f, pula se igual a zero	1 (2)	01	10bb	bfff	ffff	Nenhum
BTFSS	f,b	Testa bit b do registrador f, pula se igual a um	1 (2)	01	11bb	bfff	ffff	Nenhum
Operações de controle e com constantes								
ADDLW	k	Soma a constante k ao registrador W (W=W+k)	1	11	111x	kkkk	kkkk	C,DC,Z
ANDLW	k	AND lógico entre k e W (W=W and k)	1	11	1001	kkkk	kkkk	Z
CALL	k	Chamada da sub-rotina especificada por k	2	10	0kkk	kkkk	kkkk	Nenhum
CLRWDT		Limpa a contagem do watchdog	1	00	000	0110	0100	$\overline{TO}$, $\overline{PD}$
GOTO	k	Desvia o programa para o endereço k	2	10	1kkk	kkkk	kkkk	Nenhum
IORLW	k	OR lógico entre k e W (W=W or k)	1	11	1000	kkkk	kkkk	Z

Mnemônico	Descrição	Ciclos	Código de 14 bits MSb LSb				Flags afetados
Operações com registrador orientadas a byte							
MOVLW k	Copia a constante k para o registrador W (W=k)	1	11	00xx	kkkk	kkkk	Nenhum
NOP	Nenhuma operação	1	00	0000	0xx0	0000	Nenhum
RETFIE	Retorno da interrupção (seta GIE para 1)	2	00	0000	0000	1001	Nenhum
RETLW k	Retorna da sub-rotina, copia k para registrador W	2	11	01xx	kkkk	kkkk	Nenhum
RETURN	Retorno de sub-rotina	2	00	0000	0000	1000	Nenhum
SLEEP	Ativa modo de baixa potência	1	00	0000	0110	0011	$\overline{TO}$, $\overline{PD}$
SUBLW k	Subtrai W de k (W=k-W)	1	11	110x	kkkk	kkkk	C,DC,Z
XORLW k	XOR lógico entre k e W (W=W xor k)	1	11	1010	kkkk	kkkk	Z

C.2. 16 BITS

Mnemônico	Descrição	Ciclos	Código de 14 bits MSb LSb				Flags afetados
Operações com registrador orientadas a byte							
ADDWF f,d,a	Adiciona o valor de W ao valor de f (d=f+W)	1	0010	01da	ffff	ffff	C,DC,Z,OV,N
ADDWFC f,d,a	Adiciona o valor de W com o carry e com f	1	0010	00da	ffff	ffff	C,DC,Z,OV,N
ANDWF f,d,a	AND lógico de W com f (d=f and W)	1	0001	01da	ffff	ffff	Z,N
CLRF f,a	Limpa conteúdo de f (zera bits) (f=0)	1	0110	101a	ffff	ffff	Z
COMF f,d,a	Complementa f (inverte bits) (d= $\overline{f}$)	1	0001	11da	ffff	ffff	Z,N
CPFSEQ f,a	Compara f com W e pula se igual	1,2 ou 3	0110	001a	ffff	ffff	Nenhum
CPFSGT f,a	Compara f com W e pula se maior	1,2 ou 3	0110	010a	ffff	ffff	Nenhum
CPFSLT f,a	Compara f com W e pula se menor	1,2 ou 3	0110	000a	ffff	ffff	Nenhum
DECF f,d,a	Decrementa f (d=f-1)	1	0000	01da	ffff	ffff	C,DC,Z,OV,N
DECFSZ f,d,a	Decrementa f (d=f-1) e pula se igual a zero	1,2 ou 3	0010	11da	ffff	ffff	Nenhum
DCFSNZ f,d,a	Decrementa f (d=f-1) e pula se diferente de zero	1,2 ou 3	0100	11da	ffff	ffff	Nenhum
INCF f,d,a	Incrementa f (d=f+1)	1	0010	10da	ffff	ffff	C,DC,Z,OV,N
INCFSZ f,d,a	Incrementa f (d=f+1) e pula se igual a zero	1,2 ou 3	0011	11da	ffff	ffff	Nenhum
INFSNZ f,d,a	Incrementa f (d=f+1) e pula se diferente de zero	1,2 ou 3	0100	10da	ffff	ffff	Nenhum
IORWF f,d,a	OR lógico entre W e f (d=f or W)	1	0001	00da	ffff	ffff	Z,N
MOVF f,d,a	Copia o valor de f para o destino d (d=f)	1	0101	00da	ffff	ffff	Z,N
MOVFF f1,f2	Copia o conteúdo do registrador f1 para f2	2	1100 1111	ffff ffff	ffff ffff	ffff ffff	Nenhum
MOVWF f,a	Copia o valor de W para f (f=W)	1	0110	111a	ffff	ffff	Nenhum
MULWF f,a	Multiplica o conteúdo de W pelo conteúdo de f	1	0000	001a	ffff	ffff	Nenhum
NEGF f,a	Nega o valor de f ($f = \overline{f} + 1$)	1	0110	110a	ffff	ffff	C,DC,Z,OV,N
RLCF f,d,a	Rotaciona f uma posição à esquerda com carry	1	0011	01da	ffff	ffff	C,Z,N
RLNCF f,d,a	Rotaciona f uma posição à esquerda sem carry	1	0100	01da	ffff	ffff	Z,N
RRCF f,d,a	Desloca f uma posição à direita com carry	1	0011	00da	ffff	ffff	C,Z,N
RRNCF f,d,a	Desloca f uma posição à direita sem carry	1	0100	00da	ffff	ffff	Z,N
SETF f,a	Seta todos os bits de f	1	0110	100a	ffff	ffff	Nenhum
SUBFWB f,d,a	Subtrai f de W com empréstimo (d=W-f- $\overline{C}$)	1	0101	01da	ffff	ffff	C,DC,Z,OV,N
SUBWF f,d,a	Subtrai W de f (d=f-W)	1	0101	11da	ffff	ffff	C,DC,Z,OV,N
SUBWFB f,d,a	Subtrai W de f com empréstimo (d=f-W- $\overline{C}$)	1	0101	10da	ffff	ffff	C,DC,Z,OV,N
SWAPF f,d,a	Troca Nibbles de f (xxxxyyyy -> yyyyxxxx)	1	0011	10da	ffff	ffff	Nenhum
TSTFSZ f,a	Testa f e pula de igual a zero	1,2 ou 3	0110	011a	ffff	ffff	Nenhum
XORWF f,d,a	XOR lógico entre W e f (d=f xor W)	1	0001	10da	ffff	ffff	Z,N

Operações com registrador orientadas a bit								
BCF	f,b,a	Limpa bit b do registrador f	1	1001	bbba	ffff	ffff	Nenhum
BSF	f,b,a	Seta bit b do registrador f	1	1000	bbba	ffff	ffff	Nenhum
BTFSC	f,b,a	Testa bit b do registrador f, pula se igual a zero	1,2 ou 3	1011	bbba	ffff	ffff	Nenhum
BTFSS	f,b,a	Testa bit b do registrador f, pula se igual a um	1,2 ou 3	1010	bbba	ffff	ffff	Nenhum
BTG	f,b,a	Inverte o estado do bit b do registrador f	1	0111	bbba	ffff	ffff	Nenhum
Operações de controle e com constantes								
BC	n	Desvia se C = 1	1 (2)	1110	0010	nnnn	nnnn	nenhum
BN	n	Desvia se N = 1	1 (2)	1110	0110	nnnn	nnnn	nenhum
BNC	n	Desvia se C = 0	1 (2)	1110	0011	nnnn	nnnn	nenhum
BNN	n	Desvia se N = 0	1 (2)	1110	0111	nnnn	nnnn	nenhum
BNOV	n	Desvia se OV = 0	1 (2)	1110	0101	nnnn	nnnn	nenhum
BNZ	n	Desvia se Z = 0	1 (2)	1110	0001	nnnn	nnnn	nenhum
BOV	n	Desvia se OV = 1	1 (2)	1110	0100	nnnn	nnnn	nenhum
BRA	n	Desvia incondicionalmente	2	1101	0nnn	nnnn	nnnn	nenhum
BZ	n	Desvia se Z = 1	1 (2)	1110	0000	nnnn	nnnn	nenhum
CALL	n,s	Chamada da sub-rotina especificada por n s = 1, especifica o modo rápido	2	1110 1111	110s nnnn	nnnn nnnn	nnnn nnnn	Nenhum
CLRWDT		Limpa a contagem do watchdog	1	0000	0000	0000	0100	$\overline{TO}$, $\overline{PD}$
DAW		Ajuste decimal de W	1	0000	0000	0000	0111	C
GOTO	k	Desvia para o endereço n	2	1110 1111	1111 nnnn	nnnn nnnn	nnnn nnnn	Nenhum
NOP		Nenhuma operação	1	0000	0000	0000	0000	Nenhum
NOP		Nenhuma operação (2°. byte de outra instrução)	1	1111	xxxx	xxxx	xxxx	Nenhum
POP		Retira o conteúdo do topo da pilha e armazena no registrador TOS	1	0000	0000	0000	0110	Nenhum
PUSH		Armazena o valor de PC+2 na pilha	1	0000	0000	0000	0101	Nenhum
RCALL	k	Chamada relativa de sub-rotina	2	1101	1nnn	nnnn	nnnn	Nenhum
RESET		Resseta o processador	1	0000	0000	1111	1111	Todos
RETFIE	s	Retorna de uma interrupção, s = 1 modo rápido	2	0000	0000	0001	000s	GIE/GIEH PEIE/GIEL
RETLW	k	Retorna de uma sub-rotina com valor em W	2	0000	1100	kkkk	kkkk	Nenhum
RETURN		Retorna de uma sub-rotina	2	0000	0000	0001	001s	Nenhum
SLEEP		Coloca o chip em modo de baixo consumo	1	0000	0000	0000	0011	$\overline{TO}$, $\overline{PD}$
Operações com constantes diretas								
ADDLW	k	Soma a constante k ao registrador W (W=W+k)	1	0000	1111	kkkk	kkkk	C,DC,Z,OV,N
ANDLW	k	AND lógico entre k e W (W=W and k)	1	0000	1011	kkkk	kkkk	Z,N
IORLW	k	OR lógico entre k e W (W=W or k)	1	0000	1001	kkkk	kkkk	Z,N
LFSR	f,k	Copia a constante de 12 bits para o FSRx	2	1110 1111	1110 0000	00ff kkkk	kkkk kkkk	Nenhum
MOVLB	k	Copia a constante k para o registrador BSR	1	0000	0001	0000	kkkk	Nenhum
MOVLW	k	Copia a constante k no registrador W (W=k)	1	0000	1110	kkkk	kkkk	Nenhum
MULLW	k	Multiplica o conteúdo de W pela constante k (PROD=W*k)	1	0000	1101	kkkk	kkkk	Nenhum
RETLW	k	Retorna da sub-rotina, copia k para registrador W	2	0000	1100	kkkk	kkkk	Nenhum
SUBLW	k	Subtrai o conteúdo de W da constante k (W=k-W)	1	0000	1000	kkkk	kkkk	C,DC,Z,OV,N
XORLW	k	XOR lógico entre k e W (W=W xor k)	1	0000	1010	kkkk	kkkk	Z,N

Mnemônico	Descrição	Ciclos	Código de 14 bits MSb LSb				Flags afetados
Operações de manipulação de tabelas							
TBLRD*	Leitura de tabela	2	0000	0000	0000	1000	Nenhum
TBLRD*+	Leitura de tabela com pós-incremento	2	0000	0000	0000	1001	Nenhum
TBLRD*-	Leitura de tabela com pós-decremento	2	0000	0000	0000	1010	Nenhum
TBLRD+*	Leitura de tabela com pré-incremento	2	0000	0000	0000	1011	Nenhum
TBLWT*	Escrita de tabela	2 (5)	0000	0000	0000	1100	Nenhum
TBLWT*+	Escrita de tabela com pós-incremento	2 (5)	0000	0000	0000	1101	Nenhum
TBLWT*-	Escrita de tabela com pós-decremento	2 (5)	0000	0000	0000	1110	Nenhum
TBLWT+*	Escrita de tabela com pré-incremento	2 (5)	0000	0000	0000	1111	Nenhum

Mapas de Memória

D.1. PIC 12F675

D1.1. Banco 0

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR
00h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR. O registrador INDF não existe fisicamente								xxxx xxxx
01h	TMR0	Registrador de acesso ao conteúdo do timer 0								xxxx xxxx
02h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
03h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx
04h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
05h	GPIO	-	-	GPIO5	GPIO4	GPIO3	GPIO2	GPIO1	GPIO0	--xx xxxx
06h		Não Implementado								-
...		...								-
09h		Não Implementado								-
0Ah	PCLATH	-	-	-	Buffer de escrita dos 5 bits MSB do PC					---0 0000
0Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	GPIF	0000 0000
0Ch	PIR1	EEIF	ADIF	-	-	CMIF	-	-	TMR1IF	00-- 0--0
0Dh		Não Implementado								-
0Eh	TMR1L	Registrador de acesso ao byte menos significativo (LSB) do timer 1								xxxx xxxx
0Fh	TMR1H	Registrador de acesso ao byte mais significativo (MSB) do timer 1								xxxx xxxx
10h	T1CON	-	TMR1GE	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON	-000 0000
11h		Não Implementado								-
...		...								-
18h		Não Implementado								-
19h	CMCON	-	COUT	-	CINV	CIS	CM2	CM1	CM0	-0-0 0000
1Ah		Não Implementado								-
...		...								-
1Dh		Não Implementado								-
1Eh	ADRESH	8 bits MSB ou 2 bits MSB do resultado da conversão A/D (dependendo do modo)								xxxx xxxx
1Fh	ADCON0	ADFM	VCFG	-	-	CHS1	CHS0	$\overline{GO/DONE}$	ADON	00-- 0000

Tabela D.1 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.1.2. Banco 1

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR
80h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR. O registrador INDF não existe fisicamente								xxxx xxxx
81h	OPTION	$\overline{GPPU}$	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0	1111 1111
82h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
83h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx
84h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
85h	TRISA	-	-	TRISIO5	TRISIO4	TRISIO3	TRISIO2	TRISIO1	TRISIO0	--xx xxxx
86h		Não implementado								-
...		...								-
89h		Não implementado								-
8Ah	PCLATH	-		-		-		Buffer de escrita dos 5 bits MSB do PC		---0 0000
8Bh	INTCON	GIE	PEIE	TOIE	INTE	RBIE	TOIF	INTF	GPIF	0000 0000
8Ch	PIE1	EEIE	ADIE	-	-	CMIE	-	-	TMR1IE	00-- 0--0
8Dh		Não implementado								-
8Eh	PCON	-	-	-	-	-	-	$\overline{POR}$	$\overline{BOD}$	---- --0x
8Fh		Não implementado								
90h	OSCCAL	CAL5	CAL4	CAL3	CAL2	CAL1	CAL0	-	-	1000 00--
91h		Não implementado								-
...		....								-
94h		Não implementado								-
95h	WPU	-	-	WPU5	WPU4	WPU3	WPU2	WPU1	WPU0	--11 1111
96h	IOCB	-	-	IOCB5	IOCB4	IOCB3	IOCB2	IOCB1	IOCB0	--00 0000
97h		Não implementado								-
98h		Não implementado								-
99h	VRCON	VREN	-	VRR	-	VR3	VR2	VR1	VR0	0-0- 0000
9Ah	EEDATA	Registrador de dados da EEPROM								0000 0000
9Bh	EEADR	-	Registrador de endereçamento da EEPROM							-000 0000
9Ch	EECON1	-	-	-	-	WRERR	WREN	WR	RD	---- x000
9Dh	EECON2	Registrador de controle auxiliar da EEPROM								---- ----
9Eh	ADRESL	2 bits LSB ou 8 bits LSB do resultado da conversão A/D (dependendo do modo)								xxxx xxxx
9Fh	ANSEL	-	ADCS2	ADCS1	ADCS0	ANS3	ANS2	ANS1	ANS0	-000 1111

Tabela D.2 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.2. PIC 16F628

D.2.1. Banco 0

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR
00h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR. O registrador INDF não existe fisicamente								xxxx xxxx
01h	TMR0	Registrador de acesso ao conteúdo do timer 0								xxxx xxxx
02h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
03h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx
04h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
05h	PORTA	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0	xxxx 0000
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx
07h	-	Não implementado								-
08h	-	Não implementado								-
09h	-	Não implementado								-
0Ah	PCLATH	-	-	-	Buffer de escrita dos 5 bits MSB do PC					---0 0000
0Bh	INTCON	GIE	PEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF	0000 000x
0Ch	PIR1	EEIF	CMIF	RCIF	TXIF	-	CCP1IF	TMR2IF	TMR1IF	0000 -000
0Dh	-	Não implementado								-
0Eh	TMR1L	Registrador de acesso ao byte menos significativo (LSB) do timer 1								xxxx xxxx
0Fh	TMR1H	Registrador de acesso ao byte mais significativo (MSB) do timer 1								xxxx xxxx
10h	T1CON	-	-	T1CKPS1	TICKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON	--00 0000
11h	TMR2	Registrador de acesso ao conteúdo do timer 2								0000 0000
12h	T2CON	-	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS0	T2CKPS0	-000 0000
13h	-	Não implementado								-
14h	-	Não implementado								-
15h	CCPR1L	Registrador de Captura/Comparação/PWM (LSB)								xxxx xxxx
16h	CCPR1H	Registrador de Captura/Comparação/PWM (MSB)								xxxx xxxx
17h	CCP1CON	-	-	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000
18h	RCSTA	SPEN	RX9	SREN	CREN	ADEN	FERR	OERR	RX9D	0000 -00x
19h	TXREG	Registrador de transmissão de dados da USART								0000 0000
1Ah	RCREG	Registrador de recepção de dados da USART								0000 0000
1Bh	-	Não implementado								-
1Ch	-	Não implementado								-
1Dh	-	Não implementado								-
1Eh	-	Não implementado								-
1Fh	CMCON	C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0	0000 0000

Tabela D.3 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.2.2. Banco 1

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR
80h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR. O registrador INDF não existe fisicamente								xxxx xxxx
81h	OPTION	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111
82h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
83h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1XXX
84h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
85h	TRISA	TRISA7	TRISA6	-	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	11-1 1111
86h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111
87h	-	Não implementado								-
88h	-	Não implementado								-
89h	-	Não implementado								-
8Ah	PCLATH	-	-	-	Buffer de escrita dos 5 bits MSB do PC					---0 0000
8Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x
8Ch	PIE1	EEIE	CMIE	RCIE	TXIE	-	CCP1IE	TMR2IE	TMR1IE	0000 -000
8Dh	-	Não implementado								-
8Eh	PCON	-	-	-	-	OSCF	-	$\overline{POR}$	$\overline{BOD}$	---- 1-0x
8Fh	-	Não implementado								
90h	-	Não implementado								-
91h	-	Não implementado								-
92h	PR2	Registrador de período do timer 2								11111111
93h	-	Não implementado								-
94h	-	Não implementado								-
95h	-	Não implementado								-
96h	-	Não implementado								-
97h	-	Não implementado								-
98h	TXSTA	CSRC	TX9	TXEN	SYNC	-	BRGH	TRMT	TX9D	0000 -010
99H	SPBRG	Registrador do gerador de Baud Rate (BRG)								0000 0000
9Ah	EEDATA	Registrador de dados da EEPROM								xxxx xxxx
9Bh	EEADR	-	Registrador de endereçamento da EEPROM							xxxx xxxx
9Ch	EECON1	-	-	-	-	WRERR	WREN	WR	RD	---- x000
9Dh	EECON2	Registrador de controle auxiliar da EEPROM								---- ----
9Eh	-	Não implementado								-
9Fh	VRCON	VRE	VROE	VRR	-	VR3	VR2	VR1	VR0	000- 0000

Tabela D.4 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.2.3. Banco 2

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR
100h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR. O registrador INDF não existe fisicamente								xxxx xxxx
101h	TMR0	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111
102h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
103h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1XXX
104h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
105h	-	Não implementado								-
106h	PORT B	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111
107h	-	Não implementado								-
108h	-	Não implementado								-
109h	-	Não implementado								-
10Ah	PCLATH	-	-	-	Buffer de escrita dos 5 bits MSB do PC					---0 0000
10Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x
10Ch	-	Não implementado								-
...	-	Não implementado								-
11Fh	-	Não implementado								-

Tabela D.5 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.2.4. Banco 3

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR
180h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR. O registrador INDF não existe fisicamente								xxxx xxxx
181h	OPTION	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111
182h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
183h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1XXX
184h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
185h	-	Não implementado								-
186h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111
187h	-	Não implementado								-
188h	-	Não implementado								-
189h	-	Não implementado								-
18Ah	PCLATH	-	-	-	Buffer de escrita dos 5 bits MSB do PC					---0 0000
18Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x
18Ch	-	Não implementado								-
....	-	Não implementado								-
19Fh	-	Não implementado								-

Tabela D.6 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.3. PIC 16F876/877

D.3.1. Banco 0

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR e BOR
00h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR. O registrador INDF não existe fisicamente								xxxx xxxx
01h	TMR0	Registrador de acesso ao conteúdo do timer 0								xxxx xxxx
02h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
03h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx
04h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
05h	PORTA	-	-	RA5	RA4	RA3	RA2	RA1	RA0	xxxx 0000
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx
07h	PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	xxxx xxxx
08h	PORTD*	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	xxxx xxxx
09h	PORTE*	-	-	-	-	-	RE2	RE1	RE0	---- -xxx
0Ah	PCLATH	-	-	-	Buffer de escrita dos 5 bits MSB do PC					---0 0000
0Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x
0Ch	PIR1	PSPIF*	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000
0Dh	PIR2	-	CMIF	-	EEIF	BCLIF	-	-	CCP2IF	-0-0 0--0
0Eh	TMR1L	Registrador de acesso ao byte menos significativo (LSB) do timer 1								xxxx xxxx
0Fh	TMR1H	Registrador de acesso ao byte mais significativo (MSB) do timer 1								xxxx xxxx
10h	T1CON	-	-	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON	--00 0000
11h	TMR2	Registrador de acesso ao conteúdo do timer 2								0000 0000
12h	T2CON	-	T0UTPS3	T0UTPS2	T0UTPS1	T0UTPS0	TMR2ON	T2CKPS0	T2CKPS0	-000 0000
13h	SSPBUF	Buffer de transmissão / recepção do módulo SSP								xxxx xxxx
14h	SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	0000 0000
15h	CCPR1L	Registrador do primeiro módulo de Captura/Comparação/PWM (LSB)								xxxx xxxx
16h	CCPR1H	Registrador do primeiro módulo de Captura/Comparação/PWM (MSB)								xxxx xxxx
17h	CCP1CON	-	-	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000
18h	RCSTA	SPEN	RX9	SREN	CREN	ADEN	FERR	OERR	RX9D	0000 -00x
19h	TXREG	Registrador de transmissão de dados da USART								0000 0000
1Ah	RCREG	Registrador de recepção de dados da USART								0000 0000
1Bh	CCPR2L	Registrador do segundo módulo de Captura/Comparação/PWM (LSB)								xxxx xxxx
1Ch	CCPR2H	Registrador do segundoo módulo de Captura/Comparação/PWM (MSB)								xxxx xxxx
1Dh	CCP2CON	-	-	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000
1Eh	ADRESH	Byte mais significativo do resultado da conversão A/D								xxxx xxxx
1Fh	ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	$\overline{GO}$	-	ADON	0000 0000

Tabela D.7 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.3.2. Banco 1

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR e BOR
80h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR. O registrador INDF não existe fisicamente								xxxx xxxx
81h	OPTION	$\overline{RBPU}$	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0	1111 1111
82h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
83h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1XXX
84h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
85h	TRISA	-	-	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	--11 1111
86h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111
87h	TRISC	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0	1111 1111
88h	TRISD*	TRISD7	TRISD6	TRISD5	TRISD4	TRISD3	TRISD2	TRISD1	TRISD0	1111 1111
89h	TRISE*	IBF	OBF	IBOV	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111
8Ah	PCLATH	-	-	-	Buffer de escrita dos 5 bits MSB do PC					---0 0000
8Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x
8Ch	PIE1	PSPIE*	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000
8Dh	PIE2	-	CMIE	-	EEIE	BCLIE	-	-	CCP2IE	-0-0 0--0
8Eh	PCON	-	-	-	-	-	-	$\overline{POR}$	$\overline{BOD}$	---- --0x
8Fh	-	Não implementado								-
90h	-	Não implementado								-
91h	SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN	0000 0000
92h	PR2	Registrador de período do timer 2								1111 1111
93h	SSPADD	Registrador de endereço do módulo SSP								0000 0000
94h	SSPSTAT	SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF	0000 0000
95h	-	Não implementado								-
96h	-	Não implementado								-
97h	-	Não implementado								-
98h	TXSTA	CSRC	TX9	TXEN	SYNC	-	BRGH	TRMT	TX9D	0000 -010
99h	SPBRG	Registrador do gerador de Baud Rate (BRG)								0000 0000
9Ah	-	Não implementado								-
9Bh	-	Não implementado								-
9Ch	CMCON	C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0	0000 0111
9Dh	CVRCON	CVREN	CVROE	CVRR	-	CVR3	CVR2	CVR1	CVR0	000- 0000
9Eh	ADRESL	Byte inferior do resultado da conversão A/D								xxxx xxxx
9Fh	ADCON1	ADFM	ADCS2	-	-	PCFG3	PCFG2	PCFG1	PCFG0	0--- 0000

Tabela D.8 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.3.3. Banco 2

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR e BOR
100h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR								xxxx xxxx
101h	TMR0	$\overline{RBPU}$	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0	1111 1111
102h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
103h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1XXX
104h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
105h	-	Não implementado								-
106h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx
107h	-	Não implementado								-
108h	-	Não implementado								-
109h	-	Não implementado								-
10Ah	PCLATH	-	-	-	Buffer de escrita dos 5 bits MSB do PC					---0 0000
10Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x
10Ch	EEDATA	Byte menos significativo de acesso aos dados da EEPROM interna								xxxx xxxx
10Dh	EEADR	Byte menos significativo de endereçamento da EEPROM interna								xxxx xxxx
10Eh	EEDATH	-	-	Byte mais significativo de acesso aos dados da EEPROM						--xx xxxx
10Fh	EEADRH	-	-	-	Byte mais significativo de endereçamento da EEPROM					---x xxxx
110h	-	Não implementado								-
~	-	Não implementado								-
11Fh	-	Não implementado								-

Tabela D.9 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.3.4. Banco 3

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR e BOR
180h	INDF	Registrador de acesso indexado (indireto). Acessa o endereço especificado pelo registrador FSR								xxxx xxxx
181h	OPTION	$\overline{RBPU}$	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0	1111 1111
182h	PCL	Byte menos significativo (LSB) do contador de programa (PC)								0000 0000
183h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1XXX
184h	FSR	Ponteiro de acesso indireto à memória								xxxx xxxx
185h	-	Não implementado								-
186h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx
187h	-	Não implementado								-
188h	-	Não implementado								-
189h	-	Não implementado								-
18Ah	PCLATH	-	-	-	Buffer de escrita dos 5 bits MSB do PC					---0 0000
18Bh	INTCON	GIE	PEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF	0000 000x
18Ch	EECON1	EEPGD	-	-	-	WRERR	WREN	WR	RD	x--- x000
18Dh	EECON2	Registrador de controle auxiliar da EEPROM								---- ----
18Eh	-	Reservado, manter em 0								0000 0000
18Fh	-	Reservado, manter em 0								0000 0000
190h	-	Não implementado								-
....	-	Não implementado								-
19Fh	-	Não implementado								-

Tabela D.10 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

D.4. PIC 18F2x2/4x2

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR,BOR
F80h	PORTA	-	RA6	RA5	RA4	RA3	RA2	RA1	RA0	-x0x 0000
F81h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx
F82h	PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	xxxx xxxx
F83h	PORTD*	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	xxxx xxxx
F84h	PORTE*	-	-	-	-	-	RE2	RE1	RE0	---- -xxx
F85h	-	Não implementado								-
...		...								
F88h	-	Não implementado								-
F89h	LATA	-	LATA6	LATA5	LATA4	LATA3	LATA2	LATA1	LATA0	-xxx xxxx
F8Ah	LATB	LATB7	LATB6	LATB5	LATB4	LATB3	LATB2	LATB1	LATB0	xxxx xxxx
F8Bh	LATC	LATC7	LATC6	LATC5	LATC4	LATC3	LATC2	LATC1	LATC0	xxxx xxxx
F8Ch	LATD*	LATD7	LATD6	LATD5	LATD4	LATD3	LATD2	LATD1	LATD0	xxxx xxxx
F8Dh	LATE*	-	-	-	-	-	LATE2	LATE1	LATE0	---- -xxx
F8Eh	-	Não implementado								-
...		...								
F91h	-	Não implementado								-
F92h	TRISA	-	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	-111 1111
F93h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111
F94h	TRISC	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0	1111 1111
F95h	TRISD*	TRISD7	TRISD6	TRISD5	TRISD4	TRISD3	TRISD2	TRISD1	TRISD0	1111 1111
F96h	TRISE*	IBF	OBF	IBOV	PSPMODE	-	TRISE2	TRISE1	TRISE0	0000 -111
F97h	-	Não implementado								-
...		...								
F9Ch	-	Não implementado								-
F9Dh	PIE1	PSPIE*	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000
F9Eh	PIR1	PSPIF*	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000
F9Fh	IPR1	PSPIP*	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	1111 1111
FA0h	PIE2	-	-	-	EEIE	BCLIE	LVDIE	TMR3IE	CCP2IE	---0 0000
FA1h	PIR2	-	-	-	EEIF	BCLIF	LVDIF	TMR3IF	CCP2IF	---0 0000
FA2h	IPR2	-	-	-	EEIP	BCLIP	LVDIP	TMR3IP	CCP2IP	---1 1111
FA3h	-	Não implementado								-
FA4h	-	Não implementado								-
FA5h	-	Não implementado								-
FA6h	EECON1	EEPGD	CFGS	-	FREE	WRERR	WREN	WR	RD	xx-0 x000
FA7h	EECON2	Registrador de controle auxiliar da EEPROM interna								0000 0000
FA8h	EEDATA	Registrador de acesso aos dados da EEPROM interna								0000 0000
FA9h	EEADR	Registrador de endereçamento da memória EEPROM interna								0000 0000
FAAh	-	Não implementado								-
FABh	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x
FACh	TXSTA	CSRC	TX9	TXEN	SYNC	-	BRGH	TRMT	TX9D	0000 -010

Tabela D.11 (Continua)

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR,BOR
FADh	TXREG	Registrador de transmissão da USART								0000 0000
FAEh	RCREG	Registrador de recepção da USART								0000 0000
FAFh	SPBRG	Registrador do gerador de Baud rate								0000 0000
FB0h	-	Não implementado								-
FB1h	T3CON	RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	$\overline{\text{T3SYNC}}$	TMR3CS	TMR3ON	0000 0000
FB2h	TMR3L	Byte menos significativo da contagem do timer 3								xxxx xxxx
FB3h	TMR3H	Byte mais significativo da contagem do timer 3								xxxx xxxx
FB4h	-	Não implementado								-
...		...								
FB9h	-	Não implementado								-
FBAh	CCP2CON	-	-	DC2B1	DC2B0	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000
FBBh	CCPR2L	Byte menos significativo do registro do segundo módulo de Captura/comparação/PWM								xxxx xxxx
FBCh	CCPR2H	Byte mais significativo do registro do segundo módulo de Captura/comparação/PWM								xxxx xxxx
FBDh	CCP1CON	-	-	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000
FBEh	CCPR1L	Byte menos significativo do registro do primeiro módulo de Captura/comparação/PWM								xxxx xxxx
FBFh	CCPR1H	Byte mais significativo do registro do primeiro módulo de Captura/comparação/PWM								xxxx xxxx
FC0h	-	Não implementado								-
FC1h	ADCON1	ADFM	ADCS2	-	-	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000
FC2h	ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	-	ADON	0000 00-0
FC3h	ADRESL	Byte menos significativo do resultado da conversão A/D								xxxx xxxx
FC4h	ADRESH	Byte mais significativo do resultado da conversão A/D								xxxx xxxx
FC5h	SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN	0000 0000
FC6h	SSPCON1	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	0000 0000
FC7h	SSPSTAT	SMP	CKE	D/$\overline{\text{A}}$	P	S	R/$\overline{\text{W}}$	UA	BF	0000 0000
FC8h	SSPADD	Endereço do dispositivo no modo SSP escravo / Registrador do gerador de Baud rate no modo SSP mestre								0000 0000
FC9h	SSPBUF	Registrador do buffer de transmissão / recepção do módulo SSP								xxxx xxxx
FCAh	T2CON	-	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000
FCBh	PR2	Registrador de controle do período do timer 2								1111 1111
FCCh	TMR2	Registrador de acesso à contagem do timer 2								0000 0000
FCDh	T1CON	RD16	-	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	0-00 0000
FCEh	TMR1L	Byte menos significativo da contagem do timer 1								xxxx xxxx
FCFh	TMR1H	Byte mais significativo da contagem do timer 1								xxxx xxxx
FD0h	RCON	IPEN	-	-	$\overline{\text{RI}}$	$\overline{\text{TO}}$	$\overline{\text{PD}}$	$\overline{\text{POR}}$	$\overline{\text{BOR}}$	0-1 11qq
FD1h	WDTCON	-	-	-	-	-	-	-	SWDTE	---- ---0
FD2h	LVDCON	-	-	IRVST	LVDEN	LVDL3	LVDL2	LVDL1	LVDL0	xxxx xxxx
FD3h	OSCCON	-	-	-	-	-	-	-	SCS	---- ---0
FD4h	-	Não implementado								-
FD5h	TOCON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	1111 1111
FD6h	TMR0L	Byte menos significativo da contagem do timer 0								0000 0000
FD7h	TMR0H	Byte mais significativo da contagem do timer 0								xxxx xxxx
FD8h	STATUS	-	-	-	N	OV	Z	DC	C	---x xxxx
FD9h	FSR2L	Byte menos significativo do terceiro ponteiro de acesso indireto à memória								xxxx xxxx

Tabela D.11 (Continua)

Endereço	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor no reset POR,BOR
FDAh	FSR2H	-	-	-	-	MSB do terceiro ponteiro de acesso indireto				---- xxxx
FDBh	PLUSW2	Acesso indireto ao endereço especificado pelo FSR2 mais o conteúdo de W								-
FDCh	PREINC2	Acesso indireto ao endereço especificado pelo incremento do FSR2								-
FDDh	POSTDEC2	Acesso indireto ao endereço especificado pelo FSR2, o conteúdo do FSR2 é decrementado após a operação								-
FDEh	POSTINC2	Acesso indireto ao endereço especificado pelo FSR2, o conteúdo do FSR2 é incrementado após a operação								-
FDFh	INDF2	Acesso indireto ao endereço especificado pelo FSR2								-
FE0h	BSR	-	-	-	-	Seleção do banco de memória RAM				---- 0000
FE1h	FSR1L	Byte menos significativo do segundo ponteiro de acesso indireto a memória								xxxx xxxx
FE2h	FSR1H	-	-	-	-	MSB do segundo ponteiro de acesso indireto				---- xxxx
FE3h	PLUSW1	Acesso indireto ao endereço especificado pelo FSR1 mais o conteúdo de W								-
FE4h	PREINC1	Acesso indireto ao endereço especificado pelo incremento do FSR2								-
FE5h	POSTDEC1	Acesso indireto ao endereço especificado pelo FSR1, o conteúdo do FSR1 é decrementado após a operação								-
FE6h	POSTINC1	Acesso indireto ao endereço especificado pelo FSR1, o conteúdo do FSR1 é incrementado após a operação								-
FE7h	INDF1	Acesso indireto ao endereço especificado pelo FSR1								-
FE8h	WREG	Registrador equivalmente ao W das famílias 12,14 e 16								xxxx xxxx
FE9h	FSROL	Byte menos significativo do primeiro ponteiro de acesso indireto a memória								xxxx xxxx
FEAh	FSROH	-	-	-	-	MSB do primeiro ponteiro de acesso indireto				---- xxxx
FEBh	PLUSWO	Acesso indireto ao endereço especificado pelo FSR0 mais o conteúdo de W								-
FECh	PREINCO	Acesso indireto ao endereço especificado pelo incremento do FSR0								-
FEDh	POSTDECO	Acesso indireto ao endereço especificado pelo FSR0, o conteúdo do FSR0 é decrementado após a operação								-
FEEh	POSTINCO	Acesso indireto ao endereço especificado pelo FSR0, o conteúdo do FSR0 é incrementado após a operação								-
FEFh	INDFO	Acesso indireto ao endereço especificado pelo FSR0								-
FF0h	INTCON3	INT2IP	INT1IP	-	INT2IE	INT1IE	-	INT2IF	INT1IF	11-0 0-00
FF1h	INTCON2	RBPU	INTEDGO	INTEDG1	INTEDG2	-	TMROIP	-	RBIP	1111 -1-1
FF2h	INTCON	GIE/GIEH	PEIE/GIEL	TMROIE	INTOIE	RBIE	TMROIF	INTOIF	RBIF	0000 000x
FF3h	PRODL	Byte menos significativo do resultado de uma multiplicação								xxxx xxxx
FF4h	PRODH	Byte mais significativo do resultado de uma multiplicação								xxxx xxxx
FF5h	TABLAT	Registro de acesso ao conteúdo de uma tabela na memória de programa								0000 0000
FF6h	TBLPTRL	Byte menos significativo do ponteiro de acesso à tabela na memória de programa (<7:0>)								0000 0000
FF7h	TBLPTRH	Byte complementar do ponteiro de acesso à tabela na memória de programa (<15:8>)								0000 0000
FF8h	TBLPTRU	-	-	bit21	Byte mais significativo de acesso à tabela (< 20:16>)					--00 0000
FF9h	PCL	Byte menos significativo do contador de programa (PC) (<7:0>)								0000 0000
FFAh	PCLATH	Byte intermediário do contador de programa (PC) (<15:8>)								0000 0000
FFBh	PCLATU	-	-	-	Byte mais significativo do PC (<20:16>)					---0 0000
FFCh	STKPTR	STKFUL	STKUNF	-	Posição atual da pilha					00-0 0000
FFDh	TOSL	Byte menos significativo do conteúdo do topo da pilha (TOS <7:0>)								0000 0000
FFEh	TOSH	Byte intermediário do conteúdo do topo da pilha (TOS <15:8>)								0000 0000
FFFh	TOSU	-	-	-	Byte mais significativo do conteúdo do topo da pilha (TOS <20:16>)					---0 0000

Tabela D.11 - Não implementado (lido como '0'), u - inalterado, x - desconhecido, q - depende de condição. Os endereços sombreados não estão implementados.

Respostas dos Exercícios

Capítulo 2

1) I - d)
   II - d)
   III - b), c) e d)
   IV - c) e d)
2) d)
3) a) inteira 8 bits
   b) alfanumérica
   c) ponto flutuante ou fracionária
   d) inteira 8 bits
   e) alfanumérica
   f) booleana
   g) inteira 8 bits com sinal
   h) booleana

Capítulo 5

1) inteiro (int), caractere (char), ponto flutuante (float) e nulo (void).
2) nenhuma, mas devemos lembrar que a definição ANSI para o tipo short int é um dado inteiro de 8 bits.
3) as variáveis signed podem representar números com sinais e possuem magnitude de representação inferior às unsigned que não podem representar valores com sinais.
4) nos compiladores CCS, os tipos long int representam números de 32 bits na faixa de 0 a 4.294.967.295.
5) 127 decimal ou 01111111 em binário.
6) As variáveis globais podem ser acessadas de qualquer ponto do programa e mantêm o seu valor globalmente. As variáveis estáticas comportam-se como variáveis locais, ou seja, somente podem ser acessadas de dentro da função ou bloco de comandos em que foram declaradas. No entanto, ao contrário das variáveis locais, as estáticas não são destruídas após o encerramento da função ou bloco de comandos.
7) "teste" - global, "a" e "b" - parâmetros da função, escopo local, "valor" - variável local da função main.
8) 2
9) 0x36
10) 2
11) A modelagem de dados permite ao programador especificar o modo como o compilador vai tratar os dados envolvidos em uma expressão.

Capítulo 6

1) O = é um operador de atribuição e o == é um operador de relação de igualdade.
2) 1 ou verdadeiro.
3) 15
4) x = 1
5) 0
6) 2
7) 233
8) x = 0x34, y = 0x12

Capítulo 7

1) Verdadeira. A condição está construída de forma errada. O compilador irá avaliar a atribuição (x = 1), que é considerada verdadeira, ou seja, qualquer valor diferente de zero é considerado verdadeiro.
2) a tecla "M" maiúscula.
3) a tecla "t" seguida da tecla "b".
4) o programa imprime a tabuada do 5.

Capítulo 8

1) teste = 0x05
2) vb = 0x22
3) será o endereço da parte MSB da variável vx.
4) a) seis elementos. O último é o terminador nulo (\0).

   b) cinco elementos. O último é a letra "e'.
5) vx = 9
6) vx = 4
7) 60 bytes de memória, o valor impresso será 0,111111...

Capítulo 9

1) Não. Não podemos ter variáveis locais com o mesmo identificador de variáveis globais.
2) Declarando a função como sendo do tipo void.
3) Na passagem por valor, os valores das variáveis são passados para a função. Na passagem por referência, somente os endereços das variáveis são passados para a função.
4) São três erros: 1 - A declaração dos parâmetros está incorreta: cada variável deve ser precedida pelo seu tipo, 2 - A declaração da função está seguida pelo caractere ponto-e-vírgula, como um cabeçalho. O compilador não vai saber o que fazer com o bloco de código e gera um erro, 3 - A função foi declarada como void, mas está tentando retornar valores.
5) v1 = 5
6) V, V, F, F, F, F, V

Índice Remissivo

A

ABS, 175
ACOS, 179
Álgebra booleana, 27
Alocação de memória, 75
ANSI, 33, 54, 57, 59, 120, 138, 148
Arquivos de cabeçalho, 48
ASCII, 66
ASIN, 178
Assembler, 15
Assembly, 15, 35, 251
ASSERT, 240
ATAN, 179
ATAN2, 179
ATOF, 183
ATOI, 183
ATOI32, 183
ATOL, 183
Auto, 74

B

Bases numéricas, 67
BIT_CLEAR, 198
BIT_SET, 199
BIT_TEST, 199
Bloco de código, 49
Boolean, 59
Byte, 60

C

Caractere, 30
CAST, 71
Casting, 71
CCP, 326
CEIL, 177
CGRAM, 312, 314, 317
CGROM, 310
Char, 57
Códigos de barra invertida, 66
Comentários, 47
Compilador, 25
Comunicação, 262
  1-Wire, 282
  Assíncrona, 263
  CAN, 296
  I²C, 275
  LIN, 291
  Paralela, 262
  Serial, 262
  SPI, 268
  USART, 266
Conjunção, 27
Const, 73
Constantes, 65
Conversão de tipos, 69
Conversor
  A/D, 319
  A/D Delta - Sigma, 323
  A/D Internos, 320
COS, 179
COSH, 179
CRC, 290

D

DB9, 298
Debounce, 301
DEBUG, 24
Declaração, 61
DELAY_CYCLES, 195
DELAY_MS, 196
DELAY_US, 195
Depuração, 24
Diretivas do Compilador
  #ASM, 151
  #BIT, 152
  #BYTE, 152
  #CASE, 153
  #DEFINE, 154
  #DEVICE, 154
  #ELIF, 158, 159
  #ELSE, 158, 159
  #ENDASM, 151
  #ENDIF, 158, 159
  #ERROR, 156
  #FUSES, 156
  #ID, 157
  #IF, 158
  #IFDEF, 159
  #IFNDEF, 159
  #INCLUDE, 160
  #INLINE, 160
  #INT_AD, 161
  #INT_ADOF, 161
  #INT_BUSCOL, 161
  #INT_BUTTON, 161
  #INT_CCP1, 161
  #INT_CCP2, 161
  #INT_COMP, 161
  #INT_DEFAULT, 162
  #INT_EEPROM, 161
  #INT_EXT, 161
  #INT_EXT1, 161
  #INT_EXT2, 161
  #INT_GLOBAL, 162
  #INT_I2C, 161
  #INT_LCD, 161
  #INT_LOWVOLT, 161
  #INT_PSP, 161
  #INT_RB, 161
  #INT_RC, 161
  #INT_RDA, 161
  #INT_RTCC, 161
  #INT_SSP, 161
  #INT_TBE, 161

#INT_TIMER0, 161
#INT_TIMER1, 161
#INT_TIMER2, 161
#INT_TIMER3, 161
#LIST, 163
#LOCATE, 163
#NOLIST, 164
#OPT, 164
#ORG, 164
#PRAGMA, 166
#PRIORITY, 166
#RESERVE, 167
#ROM, 167
#SEPARATE, 168
#TYPE, 168
#UNDEF, 169
#USE DELAY, 169
#USE FAST_IO, 170
#USE FIXED_IO, 171
#USE I2C, 171
#USE RS232, 172
#USE STANDARD_IO, 170
#ZERO_RAM, 174
__DATE__, 153
__DEVICE__, 155
__FILE__, 156
__LINE__, 163
__PCB__, 165
__PCH__, 166
__PCM__, 166
__TIME__, 168
DISABLE_INTERRUPTS, 225
Disjunção, 28
Display, 308
LED 7 segmentos, 308
Displays
LCD, 308
LED, 308
Módulo LCD, 310
Double, 58
Do-While, 101, 108
Break, 109
Continue, 109
DS18S20, 285

E

E/S, 251
Eficiência, 247
ENABLE_INTERRUPTS, 224
Endereço, 120
Enumerações, 132
Escalonamento, 319
Estatísticas, 37
Estruturas, 124
Campos de Bit, 128
Declaração, 125
Operações, 126
Operador ponto, 125
Operador seta ->, 128
Ponteiros, 127
Estruturas de repetição, 101
EXP, 181
Expressões, 67
Condicionais, 68
EXT_INT_EDGE, 226
Extern, 74

F

FABS, 176
Fast_Io, 254
FGETC, 232
FGETS, 234
Fized_Io, 253
Float, 58
FLOOR, 178
Fluxogramas, 25
FMOD, 177
For, 101
Break, 105
Condição, 102
Continue, 105
Incremento, 102
Inicialização, 102
Laço Infinito, 104
Outras formas, 102
FPRINTF, 236
FPUTC, 233
FPUTS, 236
FREXP, 180
Função, 49
Funções, 137
Assembly, 147
Estruturas como argumentos, 144
Forma geral, 137
Matrizes como argumentos, 143
Parâmetros, 138
Passagem por referência, 142
Protótipos, 148
Recursão, 149
Retorno de valores, 145

G

GET_STRING, 234
GET_TIMER_X, 216
GETC, 56, 231
GETCH, 231
GETCHAR, 231
GETS, 233
Goto, 109
GOTO_ADDRESS, 229
GPIO, 301

H

HD 44780, 311

I

I/O, 251
I^2C, 241, 275
I2C_POLL, 244
I2C_READ, 242
I2C_START, 241
I2C_STOP, 241
I2C_WRITE, 243
Identificador, 51, 53, 61

If, 54, 95
Else, 96
Encadeamento, 97
INPUT, 204
INPUT_X, 205
Int, 57
Int1, 59
Int16, 60
Int32, 60
Int8, 59
Inteiros, 30
Interpretador, 25
Interrupção, 254
IOCB, 301
ISALNUM, 185
ISALPHA, 185
ISAMOUNG, 186
ISCNTRL, 185
ISDIGIT, 185
ISGRAPH, 185
ISLOWER, 185
ISPRINT, 185
ISPUNCT, 185
ISSPACE, 185
ISUPPER, 185
ISXDIGIT, 185

K

KBHIT, 238

L

LABEL_ADDRESS, 228
LABS, 175
LDEXP, 181
Linker, 74
Lista, 119
LOG, 182
LOG10, 182
Long, 59

M

Main(), 50
MAKE16, 201
MAKE32, 201
MAKE8, 200
Matrizes, 117
Adimensionais, 120
Caracteres, 122
Inicialização, 119
Inicialização por padrão, 120
Limites, 119
Multidimensionais, 121
MEMCPY, 194
MEMSET, 193
Microwire, 268
Modelagem, 71
MODF, 176
Modificadores
de armazenamento, 73
de tipo, 58
MPLAB, 43
MSB, 58

N

Negação, 29

O

OFFSETOF, 194
OFFSETOFBIT, 194
Operador
!, 82
!=, 81
%, 80
&, 83, 87
&&, 82
*, 87
?, 88
^, 84
|, 84
||, 82
~, 85
++, 80
<<, 86
==, 81
>>, 85
Abreviação, 91
Aritméticos, 80
Associação, 91
Atribuição, 79
Deslocamento, 85
Lógicos bit a bit, 83
Lógicos, 82
Memória, 87
Ponteiros, 87
Ponto, 90
Precedência, 92
Relacionais, 81
Seta ->, 90
Ternário, 88
Vírgula, 90
Otimização, 247
OUTPUT_BIT, 203
OUTPUT_FLOAT, 203
OUTPUT_HIGH, 202
OUTPUT_LOW, 202
OUTPUT_X, 205

P

Palavras reservadas, 53
Parâmetro formal, 62
Parâmetros da função, 140
Pascal, 19
PCB, 33
PCH, 33
PCM, 33
PCWH, 34
PERROR, 240
PIC
12F675, 301
16F628, 324
16F876, 303
16F877, 320, 326
16F877A, 320
Ponteiros, 113, 123

Ponto Flutuante, 30
PORT_B_PULLUPS, 206
POW, 182
PRINTF, 55, 236
PSP_INPUT_FULL, 230
PSP_OUTPUT_FULL, 230
PSP_OVERFLOW, 230
PUTC, 232
PUTCHAR, 232
PUTS, 235
PWM, 326

Q

Qualificadores de tipo, 73

R

READ_ADC, 211
READ_BANK, 227
READ_CALIBRATION, 222
READ_EEPROM, 220
READ_PROGRAM_EEPROM, 221
Redirecionamento, 134
Register, 74
RESET_CPU, 223
RESTART_CAUSE, 223
RESTART_WDT, 218
Return, 145
ROTATE_LEFT, 198
ROTATE_RIGHT, 197

S

SET_ADC_CHANNEL, 210
SET_PWMX_DUTY, 219
SET_TIMER_X, 216
SET_TRIS_X, 206
SET_UART_SPEED, 239
SETUP_ADC, 208
SETUP_ADC_PORTS, 209
SETUP_COMPARATOR, 207
SETUP_COUNTERS, 214
SETUP_PSP, 229
SETUP_RTCC, 211
SETUP_SPI, 244
SETUP_TIMER_0, 211
SETUP_TIMER_1, 212
SETUP_TIMER_2, 213
SETUP_TIMER_3, 214
SETUP_VREF, 207
SETUP_WDT, 217
SHIFT_LEFT, 197
SHIFT_RIGHT, 196
Short, 59
Signed, 58
Símbolos, 36
SIN, 178
SINH, 178
Sizeof, 91
SLEEP, 223
SPI, 244, 268
SPI_DATA_IS_IN, 246
SPI_READ, 245
SPI_WRITE, 246
SPRINTF, 193
SQRT, 183
Standard_Io, 252
STRCAT, 188
STRCHR, 189
STRCMP, 188
STRCPY, 187
STRCSPN, 191
Streams, 134
STRICMP, 188
Strings, 122
STRLEN, 186
STRLWR, 192
STRNCMP, 188
STRNCPY, 187
STRPBRK, 191
STRRCHR, 189
STRSPN, 191
STRSTR, 189
STRTOK, 190
Struct, 124
SWAP, 200
Switch, 98
    Break, 98, 100
    Case, 98
    Default, 98

T

TAN, 179
TANH, 179
Teclado, 300
Teclas, 300
Teste Condicional, 95
TIA/EIA-232, 298
TIA/EIA-485, 299
Timer 0, 258
Timer 1, 259
Timers, 260
TOLOWER, 184
TOUPPER, 184
Typecasting, 90
Typedef, 76

U

Unsigned, 59

V

Variáveis locais, 64, 140
Void, 58
Volatile, 73

W

While, 50, 55, 101, 106
    Break, 107
    Continue, 107
WPU, 301
WRITE_BANK, 227
WRITE_EEPROM, 220
WRITE_PROGRAM_EEPROM, 221

Referências Bibliográficas

GLOBO. **Microcomputador:** *Curso Prático*. São Paulo: Globo, 1987.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. **Algoritmos:** *Lógica para Desenvolvimento de Programação de Computadores*. São Paulo: Érica, 2002.

PAPPAS, C. H.; MURRAY III, W. H. **Borland C++ 4.0**. São Paulo: Makron Books, 1994.

PEREIRA, F. **Microcontroladores PIC:** *Técnicas Avançadas*. São Paulo: Érica, 2002.

PREDKO, M. **Handbook of Microcontrollers**. USA: McGraw Hill, 1998.

RITCHIE, D. M.; KERNIGHAN, B. W. **The C Programming Language**. USA: Prentice Hall, 1989.

SCHILDT, H. **C Completo e Total**. São Paulo: McGraw Hill, 1990.

_____. **Turbo C++:** *Guia do Usuário*. São Paulo: McGraw Hill, 1992.

Manuais:

CCS. **C Compiler Reference Manual**. USA: CCS, 2002.

Microchip. **PIC12F629/675 Datasheet**. USA: Microchip, 2002.

_____. **PIC16F87xA Datasheet**. USA: Microchip, 2001.

_____. **PIC18Fxx2 Datasheet**. USA: Microchip, 2001.

_____. **PICMICRO Midrange MCU Family Reference Manual**. USA: Microchip, 1997.

MOTOROLA. **M68HC11 Reference Manual**. USA: Motorola, 2001.

Sites:

- http://www.microchip.com/ - Microchip
- http://www.artimar.com.br - Representantes da Microchip no Brasil
- http://www.hitech.com.br/ - Distribuidores Microchip no Brasil
- http://www.ccsinfo.com/ - Fabricante dos compiladores CCS
- http://www.ccsinfo.com/demo.htm - Versão de demonstração do compilador PCW
- http://www.unicamp.br/fea/ortega/info/curso/fluxogrs.htm - Fluxogramas
- http://www.isi.edu/~iko/pl/hw3_c.html - C Linguage Reference
- http://www.cs.bell-labs.com/who/dmr/index.html - Página de Dennis Ritchie, criador da linguagem C
- http://www.lysator.liu.se/c/ - Programação em Linguagem C
- http://www.microchipc.com - Site sobre programação em C para PICs
- http://www.piclist.com/techref/microchip/language/c/index.htm - Página sobre programação em C da PIClist, a maior lista de discussão sobre PICs
- http://www.phanderson.com/PIC/PICC/ - Página sobre programação em C utilizando o compilador CCS

- http://www.picant.com/c2c/c.html - Página do compilador shareware C2C, inclui alguns exemplos
- http://www-s.ti.com/sc/psheets/slla037a/slla037a.pdf - Especificações do padrão 232
- http://www-s.ti.com/sc/psheets/slla070c/slla070c.pdf - Especificações do padrão 485
- http://www.mct.net/faq/spi.html - Informações sobre o protocolo SPI
- http://www.semiconductors.philips.com/buses/i2c/index.html - Especificações do protocolo I^2C
- http://pdfserv.maxim-ic.com/arpdf/AppNotes/app162.pdf - Detalhes de implementação do protocolo 1-Wire em linguagem C
- http://www.ibutton.com/ibuttons/standard.pdf - Especificações do protocolo 1-Wire
- http://pdfserv.maxim-ic.com/arpdf/DS18S20.pdf - Especificações do CI DS18S20
- http://www.lin-subbus.de/ - Especificações do protocolo LIN
- http://www.can.bosch.com/ - Especificações do protocolo CAN
- http://www.microchip.com/1010/suppdoc/appnote/all/an713/index.htm - Informações sobre o protocolo CAN
- http://pin-outs.com/directory/Hardware/PinOuts_Com/Serial_and_Parallel/ - Pinagem dos conectores mais utilizados em computadores
- http://www.microchip.com/1010/suppdoc/appnote/all/an700/index.htm - Conversor Delta - Sigma com PIC
- http://www.microchip.com/1010/suppdoc/appnote/all/an693/index.htm - Entendendo as especificações dos conversores A/D
- http://www.eletronica.etc.br/piclistbr/index.php - Versão brasileira da PICLIST, lista de discussão sobre PICs (através de e-mail)
- http://www.forumnow.com/basic/topicos.asp?grupodiscussao=13608 - Outro fórum de discussão sobre PICs (versão WEB)
- http://www.ic-prog.com - Software de programação de PICs
- http://www.symphony.com.br/ - Placas de laboratório PIC e robôs educacionais

Marcas Registradas

TRISTATE é uma marca registrada da National Semiconductors.

PIC, PICmicro, MPLAB, MPASM e ICSP são marcas registradas da Microchip Technology.

Todos os demais nomes registrados, marcas registradas, ou direitos de uso citados neste livro, pertencem aos respectivos proprietários.

MICROCONTROLADORES

PIC

PROGRAMAÇÃO EM C

O universo de programadores em linguagem C no Brasil é bastante extenso. Da mesma forma, também é grande a quantidade de utilizadores desta linguagem para a programação de microcontroladores e em especial os PICs.

A proposta deste livro é abordar a linguagem C em profundidade, mas sempre tendo como foco os microcontroladores PIC e os compiladores CCS. Num total de 12 capítulos, são apresentados os princípios básicos de programação, a linguagem C, diferenças entre C ANSI e C CCS, diretivas e funções do compilador, técnicas de programação C para PICs, técnicas de otimização, tratamento de interrupções, manipulação de E/S, manipulação de timers internos, teclados, displays (incluindo módulos LCD), comunicação serial , conversão A/D (interna e delta-sigma), PWM, etc. Os exemplos são baseados nos principais PICs disponíveis atualmente: 12F675, 16F62x, 16F87x e 18Fxx2.

No tópico de comunicação serial, é apresentado um estudo aprofundado dos protocolos: assíncrono (vulgo RS-232), SPI, I2C, 1-Wire, LIN e CAN e diversas bibliotecas C para implementação dos protocolos de comunicação.

O livro conta ainda com diversos exercícios de fixação e exemplos, tais como: terminal RS-232 com LCD, conversores A/D com comunicação serial, comunicação com memórias I2C, controle de brilho de led com PWM, medição de temperatura com DS18S20, teclado de 12 teclas com auto-repetição, animação em LCD com caracteres definidos pelo usuário, etc.

Ao final há um conjunto de apêndices com informações como: tabela ASCII, sumário de funções C, mapas de memória dos PIC utilizados, conjuntos de instruções de 14 e 16 bits, etc.

Enfim, trata-se de um livro indispensável tanto ao aprendizado de C como à utilização no dia-a-dia do programador.

ISBN: 85-7194-935-2
9 788571 949355

EDITORA ÉRICA LTDA.
Rua São Gil, 159 • Tatuapé • CEP: 03401-030 • São Paulo • SP
Fone: (11) 295-3066 • Fax: (11) 6197-4060
www.editoraerica.com.br

Invista em você.
Visite uma livraria.

MICROCONTROLADORES

FÁBIO PEREIRA

PIC

PROGRAMAÇÃO EM C

2ª EDIÇÃO

Baseado nos compiladores CCS

Exemplos com:
12F675, 16F62x, 16F87x e 18Fxx2

Inclui bibliotecas para
comunicação assíncrona, SPI, I2C,
1-Wire, manipulação de LCD e outros

BRINDE
Arquivos de exemplo e bibliotecas apresentados no capítulo 12 e arquivos de cabeçalho com as definições de registradores utilizados nos programas do livro disponíveis na INTERNET.