

AVR e Arduino

Técnicas de Projeto

Charles Borges de Lima

e

Marco V. M. Villaça

AVR e Arduino
Técnicas de Projeto

2^a edição

Florianópolis
Edição dos Autores
2012

Copyright © 2012 para Charles Borges de Lima e Marco Valério Miorim Villaça.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, dos autores.

Figura da capa de www.dreamstime.com.

L732a Lima, Charles Borges de
AVR e Arduino : técnicas de projeto / Charles Borges de Lima,
Marco Valério Miorim Villaça. 2. ed. – Florianópolis: Ed. dos autores,
2012.
632 p.: il; 21,0 cm.

Inclui bibliografia
ISBN: 978-85-911400-1-5

1. Microcontroladores – AVR. 2. Arduino. 3. Engenharia eletrônica.
4. Projetos eletrônicos. 5. Programação. I Villaça, Marco Valério
Miorim. II. Título.

CDD:
621.381

Sistema de Bibliotecas Integradas do IFSC.
Biblioteca Dr. Hercílio Luz – Campus Florianópolis.
Catalogado por: Rose Mari Lobo Goulart, CRB 14/277.

Todas as marcas registradas, nomes ou direitos de uso citados neste livro pertencem aos seus respectivos proprietários.

As técnicas de projetos eletrônicos apresentadas não são exclusivas, são técnicas comuns empregadas com microcontroladores. Elas estão descritas em referências técnicas, sendo informações empregadas no meio industrial e acadêmico.

Os autores acreditam que todas as informações aqui apresentadas estão corretas e podem ser utilizadas para qualquer fim legal. Entretanto, não existe qualquer garantia implícita ou explícita, de que o uso de tais informações conduzirá sempre ao resultado esperado. Os nomes de sítios 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.

Os programas e materiais suplementares estão disponíveis para download em:
www.borgescorporation.blogspot.com

Eventualmente, uma errata poderá ser disponibilizada.

Junho/2012.

Charles dedica:

À minha querida esposa Gisela e a minha pequena Cecília, luzes do meu caminho.

Marco dedica:

À minha querida esposa Tade-Ane e aos meus três especiais filhos, a certeza da continuação da minha existência, Caio, Pedro e Vicente.

“Não queremos apenas deixar um mundo melhor para nossos filhos, queremos muito mais deixar filhos melhores para o mundo”.

AGRADECIMENTOS

É com sinceridade que agradecemos a todas as pessoas que colaboraram para o desenvolvimento desta obra:

- Aos colegas do Departamento de Eletrônica do Instituto Federal de Santa Catarina, pelo suporte moral e técnico.
- Aos alunos, sem os quais esta obra não seria possível, tornando visível, com suas necessidades e sugestões, o caminho a seguir.
- Aos amigos, onde a expressão da palavra e dos atos é verdadeira.
- Por último, mas não menos importante: especialmente as nossas esposas e filhos, pelo apoio em todos os momentos e pela compreensão das horas que abdicamos de suas companhias (que não foram poucas), na escrita e desenvolvimento do texto e projetos apresentados.

O DISCÍPULO E O BALAIO

Um discípulo chegou para seu mestre e perguntou:

- Mestre, por que devemos ler e decorar tantas orações se não conseguimos memorizá-las completamente e com o tempo as esquecemos?

O mestre não respondeu imediatamente. Ele ficou olhando para o horizonte e depois ordenou ao discípulo:

- Pegue aquele balaio de juncos, desça até o riacho, o encha de água e o traga até aqui.

O discípulo olhou para o balaio, que estava bem sujo, e achou muito estranha a ordem do mestre, mas mesmo assim, obedeceu. Pegou o balaio sujo, desceu os 100 degraus da escadaria até o riacho, encheu o cesto de água e começou a subir de volta.

Como o balaio era todo cheio de furos, a água foi escorrendo e quando o discípulo chegou até o mestre, já não restava mais água nenhuma.

O mestre, então, perguntou:

- Então, meu filho, o que você aprendeu?

O discípulo olhou para o cesto vazio e disse:

- Aprendi que um balaio de juncos não segura água.

O mestre ordenou-lhe que repetisse o processo.

Quando o discípulo voltou com o balaio vazio novamente, o mestre perguntou:

- Então, meu filho, e agora, o que você aprendeu?

O discípulo novamente respondeu com sarcasmo:

- Balaio furado não segura água.

O mestre, então, continuou ordenando que o discípulo repetisse a tarefa.

Depois da décima vez, o discípulo estava todo molhado e exausto de tanto descer e subir as escadas. Porém, quando o mestre perguntou de novo:

- Então, meu filho, o que você aprendeu?

O discípulo, olhando para dentro do balaio, percebeu admirado:

- O balaio está limpo! Apesar de não segurar a água, ela acabou por lavá-lo!

O mestre, por fim, concluiu:

- Então meu filho, não importa que você não consiga decorar todas as orações.

O que importa, na verdade, é que elas purificam sua mente e sua alma.

Antiga história de um mosteiro chinês.

- O que adianta estudar e estudar, se muito do que estudamos acaba por ser esquecido?

- O estudo melhora o nosso processo cognitivo e nosso raciocínio lógico, preparando-nos mais e mais para enfrentar os problemas da vida. Portanto, comece a passar mais água no seu balaio.

SUMÁRIO

PREFÁCIO	XVII
1. INTRODUÇÃO	1
1.1 ATUALIDADE	1
1.2 MICROPROCESSADORES	3
1.3 TIPOS DE MEMÓRIAS	8
1.4 MICROCONTROLADORES	9
1.5 OS MICROCONTROLADORES AVR	12
1.6 A FAMÍLIA AVR	14
2. O ATMEGA328	17
2.1 AS MEMÓRIAS.....	22
2.1.1 O REGISTRADOR DE STATUS - SREG	26
2.1.2 O STACK POINTER	28
2.2 DESCRIÇÃO DOS PINOS	30
2.3 SISTEMA DE CLOCK	33
2.4 O RESET	34
2.5 GERENCIAMENTO DE ENERGIA E O MODO SLEEP	37
3. SOFTWARE E HARDWARE	39
3.1 A PLATAFORMA ARDUINO	40
3.2 CRIANDO UM PROJETO NO AVR STUDIO	43
3.3 SIMULANDO NO PROTEUS (ISIS)	49
3.4 GRAVANDO O ARDUINO	52
4. PROGRAMAÇÃO	57
4.1 ALGORITMOS	58
4.1.1 ESTRUTURAS DOS ALGORITMOS	60
4.2 FLUXOGRAMAS	61
4.3 MÁQUINA DE ESTADOS.....	63
4.4 O ASSEMBLY	64
4.5 PROGRAMAÇÃO C	67
4.5.1 INTRODUÇÃO	69

4.5.2 ESTRUTURAS CONDICIONAIS.....	81
4.5.3 ESTRUTURAS DE REPETIÇÃO	83
4.5.4 DIRETIVAS DE PRÉ-COMPILAÇÃO	87
4.5.5 PONTEIROS	91
4.5.6 VETORES E MATRIZES	91
4.5.7 ESTRUTURAS.....	92
4.5.8 FUNÇÕES	93
4.5.9 O IMPORTANTÍSSIMO TRABALHO COM BITS	95
4.5.10 PRODUZINDO UM CÓDIGO EFICIENTE	101
4.5.11 ESTRUTURAÇÃO E ORGANIZAÇÃO DO PROGRAMA.....	102
5. PORTAS DE ENTRADA E SAÍDA (I/OS).....	105
5.1 ROTINAS SIMPLES DE ATRASO	107
5.2 LIGANDO UM LED	110
5.3 LENDO UM BOTÃO (CHAVE TÁCTIL).....	114
5.4 ACIONANDO DISPLAYS DE 7 SEGMENTOS	125
5.5 ACIONANDO LCDs 16 x 2	132
5.5.1 INTERFACE DE DADOS DE 8 BITS	133
5.5.2 INTERFACE DE DADOS DE 4 BITS	137
5.5.3 CRIANDO NOVOS CARACTERES	146
5.5.4 NÚMEROS GRANDES	149
5.6 ROTINAS DE DECODIFICAÇÃO PARA USO EM DISPLAYS.....	154
6. INTERRUPÇÕES	157
6.1 INTERRUPÇÕES NO ATMEGA328	157
6.2 INTERRUPÇÕES EXTERNAS.....	163
6.2.1 UMA INTERRUPÇÃO INTERROMPENDO OUTRA.....	170
7. GRAVANDO A EEPROM	173
8. TECLADO MATRICIAL.....	179
9. TEMPORIZADORES/CONTADORES.....	185
9.1 TEMPORIZANDO E CONTANDO	186
9.2 MODULAÇÃO POR LARGURA DE PULSO – PWM	187
9.3 TEMPORIZADOR/CONTADOR 0	189
9.3.1 REGISTRADORES DO TCO	194

9.3.2 CÓDIGOS EXEMPLO	199
9.4 TEMPORIZADOR/CONTADOR 2.....	202
9.4.1 REGISTRADORES DO TC2	203
9.4.2 CÓDIGO EXEMPLO	208
9.5 TEMPORIZADOR/CONTADOR 1.....	211
9.5.1 REGISTRADORES DO TC1	217
9.5.2 CÓDIGOS EXEMPLO	221
9.6 PWM POR SOFTWARE.....	226
9.7 ACIONANDO MOTORES	228
9.7.1 MOTOR DC DE IMÃ PERMANENTE	228
9.7.2 MICRO SERVO MOTOR	231
9.7.3 MOTOR DE PASSO UNIPOLAR.....	234
9.8 MÓDULO DE ULTRASSOM - SONAR.....	238
10. GERANDO MÚSICAS COM O MICROCONTROLADOR	243
11. TÉCNICAS DE MULTIPLEXAÇÃO.....	253
11.1 EXPANSÃO DE I/O MAPEADA EM MEMÓRIA	254
11.2 CONVERSÃO SERIAL-PARALELO	257
11.3 CONVERSÃO PARALELO-SERIAL	260
11.4 MULTIPLEXAÇÃO DE DISPLAYS DE 7 SEGMENTOS	260
11.5 UTILIZANDO MAIS DE UM LED POR PINO	266
11.6 MATRIZ DE LEDs	268
11.7 CUBO DE LEDs	271
12. DISPLAY GRÁFICO (128 × 64 PONTOS)	277
12.1 CONVERSÃO DE FIGURAS.....	283
12.2 USANDO O DISPLAY GRÁFICO COMO UM LCD 21 x 8	293
13. FORMAS DE ONDA E SINAIS ANALÓGICOS.....	297
13.1 DISCRETIZANDO UM SINAL	297
13.2 CONVERSOR DIGITAL-ANALÓGICO COM UMA REDE R/2R.....	302
13.3 CONVERSOR DIGITAL-ANALÓGICO COM UM SINAL PWM	305
13.4 SINAL PWM PARA UM CONVERSOR CC-CC (BUCK)	307
13.5 SINAL PWM PARA UM CONVERSOR CC-CA	310
14. SPI	317

14.1 SPI DO ATMEGA328	320
14.2 SENSOR DE TEMPERATURA TC72	326
14.3 CARTÃO DE MEMÓRIA SD/MMC	331
15. USART.....	345
15.1 USART DO ATMEGA328	345
15.2 USART NO MODO SPI MESTRE	357
15.3 COMUNICAÇÃO ENTRE O MICROCONTROLADOR E UM COMPUTADOR	358
15.4 MÓDULOS BÁSICOS DE RF	367
15.5 MÓDULO BLUETOOTH PARA PORTA SERIAL	373
16. TWI (TWO WIRE SERIAL INTERFACE) – I2C	377
16.1 I2C NO ATMEGA328.....	379
16.2 REGISTRADORES DO TWI	383
16.3 USANDO O TWI	386
16.4 RELÓGIO DE TEMPO REAL DS1307	396
17. COMUNICAÇÃO 1 FIO POR SOFTWARE	411
17.1 SENSOR DE TEMPERATURA DS18S20.....	415
18. COMPARADOR ANALÓGICO	425
18.1 MEDINDO RESISTÊNCIAS E CAPACITÂNCIAS	430
18.2 LENDO MÚLTIPLAS TECLAS	434
19. CONVERSOR ANALÓGICO-DIGITAL - ADC	437
19.1 REGISTRADORES DO ADC.....	444
19.2 MEDAÇÃO DE TEMPERATURA.....	448
19.3 LENDO UM TECLADO MATRICIAL.....	451
19.4 SENSOR DE TEMPERATURA LM35.....	453
19.5 SOBREAMOSTRAGEM	456
19.6 FILTRO DE MÉDIA MÓVEL	459
20. MESCLANDO PROGRAMAÇÃO C COM ASSEMBLY	463
20.1 USO DOS REGISTRADORES DO AVR	465
20.2 CONVENÇÃO NA CHAMADA DE FUNÇÕES	465
20.3 EXEMPLOS	466
20.4 O ARQUIVO MAP	470

21. RTOS.....	473
21.1 INTRODUÇÃO	473
21.2 GESTÃO DE TAREFAS	477
21.3 COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE TAREFAS.....	483
21.4 ALOCAÇÃO DINÂMICA DE MEMÓRIA.....	486
21.5 BRTOS	488
22. A IDE DO ARDUINO	499
23. FUSÍVEIS E A GRAVAÇÃO DO ATMEGA328.....	507
23.1 OS FUSÍVEIS E BITS DE BLOQUEIO	508
23.2 HARDWARE DE GRAVAÇÃO	512
23.3 O TEMPORIZADOR WATCHDOG.....	515
23.4 GRAVAÇÃO IN-SYSTEM DO ATMEGA	520
24. CONSIDERAÇÕES FINAIS.....	523
25. REFERÊNCIAS	529

APÊNDICE

A. ASSEMBLY DO ATMEGA.....	537
A.1 INSTRUÇÕES DO ATMEGA	537
A.2 STATUS REGISTER	542
B. DISPLAY DE CRISTAL LÍQUIDO 16 × 2 - CONTROLADOR HD44780	543
B.1 PINAGEM	543
B.2 CÓDIGOS DE INSTRUÇÕES	544
B.3 ENDEREÇO DOS SEGMENTOS (DDRAM)	545
B.4 CONJUNTO E CÓDIGO DOS CARACTERES.....	545
C. ERROS E AJUSTE DA TAXA DE COMUNICAÇÃO DA USART	547
D. CIRCUITOS PARA O ACIONAMENTO DE CARGAS	551

D.1 A CHAVE TRANSISTORIZADA	551
D.2 CIRCUITOS INTEGRADOS PARA O SUPRIMENTO DE CORRENTES	558
D.3 ACOPLADORES ÓPTICOS	560
E. PROJETO DE PLACAS DE CIRCUITO IMPRESSO - BÁSICO.....	563
E.1 UNIDADE IMPERIAL E MÉTRICA	563
E.2 ENCAPSULAMENTOS	564
E.2.1 COMPONENTES PTH	565
E.2.2 COMPONENTES SMD.....	567
E.3 PADS E VIAS (PTH)	568
E.4 TRILHAS.....	569
E.5 REGRAS BÁSICAS DE DESENHO	571
E.6 PASSOS PARA O DESENHO DA PCI.....	575
E.7 CONCLUSÕES	581
F. TABELAS DE CONVERSÃO	583
G. REGISTRADORES DE I/O DO ATMEGA328.....	587
H. ACRÔNIMOS.....	609

PREFÁCIO

Aristóteles em sua Metafísica falou que "na verdade, foi pelo espanto que os homens começaram a filosofar tanto no princípio como agora". Todo aquele que inicia o aprendizado dos microcontroladores e se espanta com todas as possibilidades de uso da tecnologia, certamente se tornará um eterno e apaixonado estudante desses dispositivos. A constante evolução dos microcontroladores corrobora os ensinamentos filosóficos de Heráclito de Éfeso, segundo o qual tudo é devir - "não entramos duas vezes em um mesmo rio". A todo instante, somos surpreendidos com o lançamento de novos dispositivos, que superam os seus antecessores em capacidade de processamento e no conjunto de suas aplicações; acompanhar esses avanços requer a disposição para um aprendizado contínuo.

Em 1980, quando a Intel lançou o legendário microcontrolador 8051, que operava a 12 MHz e embarcava 4 kbytes de ROM para receber o código do usuário e 128 bytes de RAM para armazenar dados temporários, não se podia imaginar que uma dezena de microprocessadores ou microcontroladores estariam presentes em nossos lares e carros, na forma de relógios, brinquedos, eletrodomésticos, telefones, computadores de bordo, etc. Fazemos referência ao 8051, porque foi através desse simpático senhor de 32 anos que a nossa geração de técnicos e engenheiros realizaram seus primeiros projetos com microcontroladores. E, ao ingressar na carreira docente, trabalhávamos os conceitos de programação e hardware de sistemas embarcados utilizando a arquitetura MCS-51. Mais tarde, devido à compatibilidade pino a pino entre o 8051 e um dos microcontroladores da família AVR, à época a arquitetura de 8 bits de melhor desempenho, optamos em trabalhar nas disciplinas de microcontroladores com os dispositivos AVR. Com uma pequena modificação na nossa placa desenvolvimento que empregava um microcontrolador AT89S8252, um 8051 da Atmel, começamos a utilizar o

AT90S8515 e, posteriormente, o ATmega8515. Depois, com adaptações e melhorias para o ensino, acabamos desenvolvendo um kit com o ATmega32, incluindo inúmeros periféricos. Atualmente, utilizamos a plataforma Arduino com o ATmega328, auxiliados pelo seu amplo conjunto de módulos de expansão de hardware.

Lançado no ano que o Curso Técnico de Eletrônica do Departamento Acadêmico de Eletrônica do Instituto Federal de Santa Catarina - Campus Florianópolis comemora seu jubileu de prata, este livro, de certa forma, materializa parte das discussões e conhecimentos acumulados no nosso departamento ao longo destes anos e preenche uma lacuna técnica, em língua portuguesa, ao apresentar um grande conjunto de técnicas de projetos com microcontroladores reunidas de forma didática em um único compêndio.

Neste livro, para a descrição do AVR, optou-se pelo Atmega328, que apresenta as principais características desejáveis em um microcontrolador e, ainda, está presente no Arduino Uno, a versão mais popular do módulo de desenvolvimento de código aberto Arduino, plataforma de fácil utilização, inclusive por leigos, e que conta ainda com uma intensa e ampla comunidade global que troca informações e dissemina o seu uso.

A grande maioria dos programas foi testada no Arduino Uno com a utilização de hardware complementar na forma de módulos prontos ou montados em matriz de contato. Como o Arduino é uma plataforma de desenvolvimento fácil de encontrar e barata, possuindo amplo hardware de suporte, fica fácil testar os programas apresentados, permitindo também o rápido desenvolvimento de novos projetos. Os programas foram desenvolvidos no AVR Studio® 5.1, o software profissional e gratuito disponibilizado pela Atmel.

Também apresentamos um software para a simulação de microcontroladores (Proteus®), cada vez mais utilizado no meio acadêmico e na indústria, superando a velha prerrogativa do teste do programa em hardware.

Para conhecer os detalhes do microcontrolador, é fundamental a consulta ao manual do fabricante. Aqui apresentamos apenas um resumo e algumas aplicações típicas, ressaltando que as figuras e detalhes técnicos do ATmega foram obtidos desse manual. É importante notar que o fabricante disponibiliza vários *Application Notes* com muitas dicas de projeto, bem como códigos prontos para serem utilizados.

Desenvolvemos os conteúdos para serem ministrados durante um semestre letivo, em cursos com não menos que 4 horas semanais. Entretanto, o conteúdo pode ser dividido em dois semestres de acordo com a carga horária empregada e de acordo com a profundidade no desenvolvimento dos temas apresentados. Isso só é valido para alunos que possuem conhecimento técnico em eletrônica e programação. Todavia, tal conhecimento não é prerrogativa para o estudo deste livro, mas fundamental para a compreensão de muito dos conceitos apresentados.

São propostos inúmeros exercícios ao longo dos capítulos. Eles servem para fixar conceitos e permitir ao aluno aprender novas técnicas, incluindo sugestões de projetos e considerações técnicas. Apesar de ser possível exclusivamente simular os circuitos, o desenvolvimento em hardware não deve ser desprezado, permitindo a conexão entre teoria e prática, aumentando o aprendizado, a capacidade lógica de projeto e a solução de problemas.

Tentamos apresentar os conceitos de forma objetiva para a leitura rápida e fácil. Desta forma, os esquemáticos dos circuitos foram simplificados para facilitar a visualização e compreensão. A complexidade e o conhecimento são gradualmente aumentados ao longo do texto. Após a leitura dos capítulos introdutórios é importante seguir os capítulos ordenadamente. Entretanto, os capítulos de conhecimentos gerais sobre os softwares empregados (capítulo 3) e sobre programação (capítulo 4) seguem uma sequência para melhor organização do conteúdo, podendo ser consultados sempre que necessário.

No capítulo 4 apresentamos uma breve revisão sobre as técnicas necessárias para a programação a fim de suprir eventuais dúvidas e orientar o estudante para a programação microcontrolada. Nesse capítulo, também apresentamos sugestões para o emprego eficiente da linguagem C para o ATmega, conforme recomendado pela Atmel, e explicações para o importantíssimo trabalho com bits.

Quanto à linguagem de programação *assembly*, seu emprego é fundamental para a compreensão da estrutura interna do microcontrolador e sua forma de funcionamento. Pode-se aprender muito com os códigos exemplo apresentados. Todavia, a linguagem C é a principal forma de programação, dada sua portabilidade e facilidade de uso, independente do tipo de microcontrolador empregado. Programar em *assembly* só deve ser considerado para fins de estudo ou quando realmente houver limitações de desempenho e memória do microcontrolador a considerar, por isso empregamos muito pouco o *assembly* e concentramos o foco na programação em linguagem C. Porém, existe um capítulo exclusivo de como utilizar ambos, C e *assembly*, em um único programa, explorando o que há de melhor nas duas linguagens.

No apêndice apresentamos informações relevantes e necessárias para a programação e desenvolvimento de circuitos microcontrolados. Entre elas, estão dois capítulos: um sobre o projeto de chaves transistorizadas, o outro sobre técnicas de desenho para placas de circuito impresso. É importante um contato com esse material quando do projeto completo de circuitos microcontrolados.

Em resumo, este livro é um ponto inicial no desenvolvimento de projetos microcontrolados, cujas técnicas são empregadas para o ensino e projeto. Esperamos, realmente, que ele possa ajudá-lo e inspirá-lo quanto às possibilidades de projetos com microcontroladores.

Bom Estudo!

Os autores.

1. INTRODUÇÃO

Neste capítulo, serão abordados alguns conceitos necessários à compreensão dos microcontroladores, começando com os microprocessadores e as memórias. Posteriormente, será feita uma introdução aos microcontroladores e a apresentação dos mais relevantes disponíveis no mercado, incluindo detalhes sobre os AVR.

1.1 ATUALIDADE

As melhorias tecnológicas demandam cada vez mais dispositivos eletrônicos. Assim, a cada dia são criados componentes eletrônicos mais versáteis e poderosos. Nessa categoria, os microprocessadores, aos quais os microcontroladores pertencem, têm alcançado grande desenvolvimento. Sua facilidade de uso em amplas faixas de aplicações permite o projeto relativamente rápido e fácil de novos equipamentos. Atualmente, os microcontroladores estão presentes em quase todos os dispositivos eletrônicos controlados digitalmente, como por exemplo: nas casas, em máquinas de lavar, fornos de micro-ondas, televisores, aparelhos de som e imagem, condicionadores de ar e telefones; nos veículos, em sistemas eletrônicos de controle de injeção de combustível, controle de estabilidade, freios ABS (*Anti-lock Braking System*), computadores de bordo e GPS (*Global Positioning System*); nos eletrônicos portáteis, em telefones celulares, tocadores de mídia eletrônica, video games e relógios; na indústria, em controladores lógico programáveis, de motores e fontes de alimentação. Em resumo, os sistemas microcontrolados encontram-se em todos os segmentos eletrônicos, desde simples equipamentos domésticos até sistemas industriais complexos (fig. 1.1).

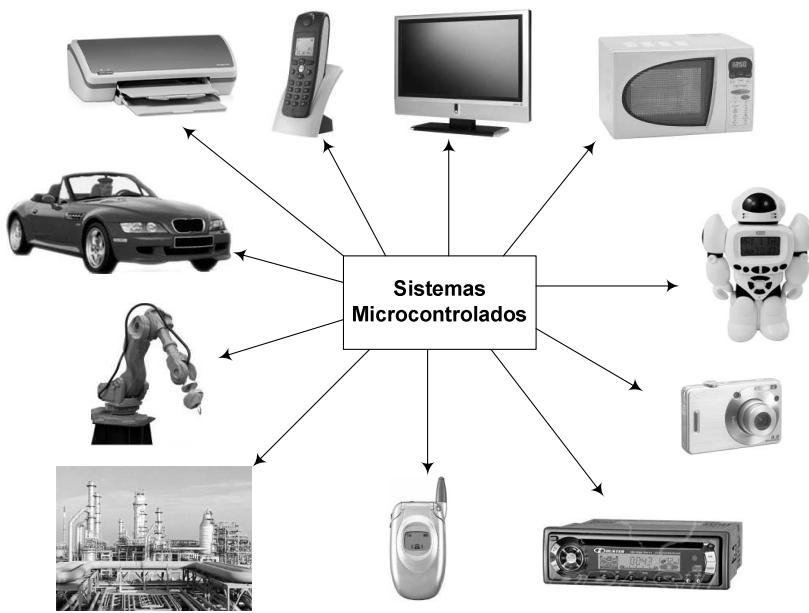


Fig. 1.1 - Sistemas microcontrolados.

O desenvolvimento dos microcontroladores se deve ao grande número de funcionalidades disponíveis em um único circuito integrado. Como o seu funcionamento é ditado por um programa, a flexibilidade de projeto e de formas de trabalho com um hardware específico são inúmeras, permitindo aplicações nas mais diversas áreas.

1.2 MICROPROCESSADORES

Com a evolução da microeletrônica, o nível de miniaturização e o número de funcionalidades dos circuitos integrados evoluíram, permitindo o desenvolvimento de circuitos compactos e programáveis. Assim, surgiram os primeiros microprocessadores.

Um microprocessador é um circuito integrado composto por inúmeras portas lógicas, organizadas de tal forma que permitem a realização de operações digitais, lógicas e aritméticas. Sua operação é baseada na decodificação de instruções armazenadas em uma memória programável. A execução das instruções é ditada por um sinal periódico (relógio - *clock*).

Um diagrama básico de um sistema microprocessado é apresentado na fig. 1.2. O oscilador é responsável pela geração do sinal síncrono de *clock*, o qual faz com que o microprocessador funcione. O coração do sistema é a Unidade Central de Processamento (CPU – *Central Processing Unit*), a qual realiza as operações lógicas e aritméticas exigidas pelo programa. O processador dispõe de duas memórias: uma, onde está o código de programa a ser executado (externa ao chip); e outra, temporária, onde são armazenados informações para o trabalho da CPU (memória de dados). A CPU pode receber e enviar informações ao exterior através da sua interface de entrada e saída. As informações dentro do microprocessador transitam por barramentos, cuja estrutura depende da arquitetura empregada.

Para ler a memória de programa, a CPU possui um contador de endereços (PC – *Program Counter*) que indica qual posição da memória deve ser lida. Após a leitura do código no endereço especificado, o mesmo é decodificado e a operação exigida é realizada. O processo se repete indefinidamente de acordo com o programa escrito.

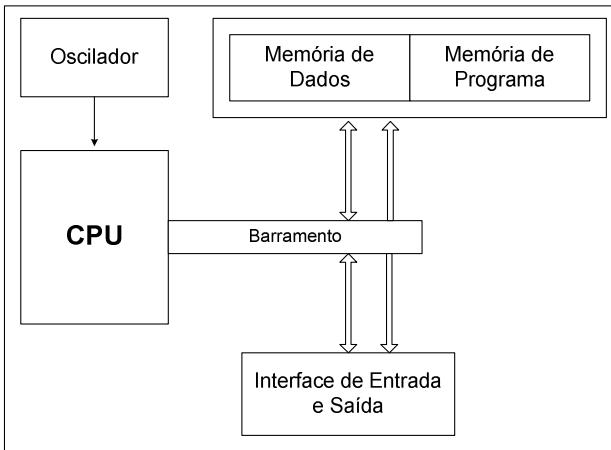


Fig. 1.2 – Estrutura básica de um sistema microprocessado.

Quanto à organização do barramento, existem duas arquiteturas predominantes para as CPUs dos microprocessadores: a arquitetura Von-Neumann, onde existe apenas um barramento interno por onde circulam dados e instruções, e a arquitetura Harvard, caracterizada por dois barramentos internos, um de instruções e outro de dados (Fig. 1.3). Na arquitetura Von-Neumann, a busca de dados e instruções não pode ser executada ao mesmo tempo (gargalo da arquitetura), limitação que pode ser superada com a busca antecipada de instruções (*pipeline*¹) e/ou com caches de instruções/dados. Já na arquitetura Harvard, os dados e instruções podem ser acessados simultaneamente, o que torna essa arquitetura inherentemente mais rápida que a Von-Neumann.

Por sua vez, quanto ao conjunto de instruções, os microprocessadores são classificados em duas arquiteturas: Computadores com Conjunto Complexo de Instruções (CISC – *Complex Instructions Set Computers*) e Computadores com Conjunto Reduzido de Instruções (RISC – *Reduced*

¹ **Pipeline** é uma técnica de hardware que permite a execução em paralelo de operações por parte da CPU. As instruções são colocadas em um fila aguardando o momento de sua execução. Assim, pode-se imaginar um sistema com registradores organizados em cascata e cujo funcionamento é sincronizado por um sinal de *clock*.

Instructions Set Computers). A arquitetura RISC utiliza um conjunto de instruções simples, pequeno e geralmente com extensão fixa. Semelhante a RISC, a arquitetura CISC utiliza um conjunto simples de instruções, porém utiliza também instruções mais longas e complexas, semelhantes às de alto nível, na elaboração de um programa. As instruções CISC são geralmente variáveis em extensão. Assim, a arquitetura RISC necessita de mais linhas de código para executar a mesma tarefa que uma arquitetura CISC, a qual possui muito mais instruções.

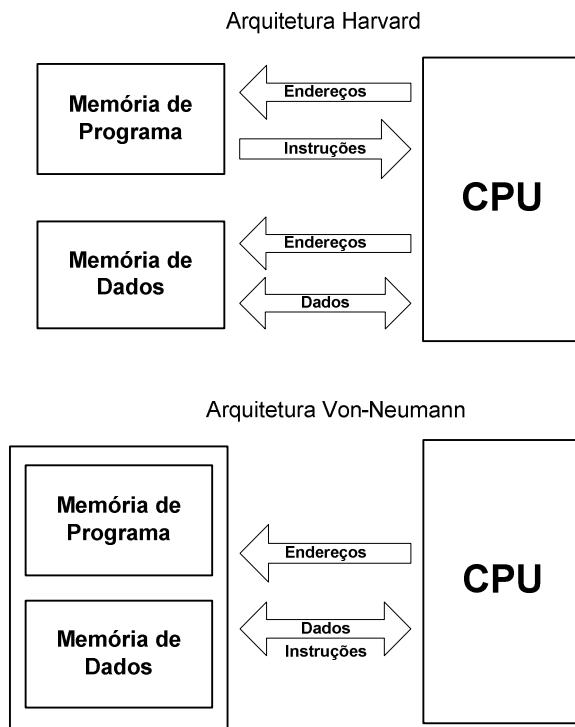


Fig. 1.3 – Arquiteturas clássicas de microprocessadores: Harvard × Von-Neumann.

Muitas literaturas associam a arquitetura Von-Neumann com a arquitetura CISC e a arquitetura Harvard com a arquitetura RISC. A provável razão de tal associação é o fato de muitos microprocessadores

com arquitetura Von-Neumann serem CISC e muitos microprocessadores Harvard serem RISC. Entretanto, a arquitetura CISC e RISC não são sinônimos para as arquiteturas Von-Neumann e Harvard, respectivamente. Por exemplo:

- O núcleo ARM7 é Von-Neumann e RISC.
- O PowerPC é um microprocessador RISC puro, Harvard internamente e Von-Neumann externamente.
- O Pentium-Pro é um microprocessador RISC e Harvard internamente e Von-Neumann e CISC externamente.

A fig. 1.4 é empregada para exemplificar a diferença entre as duas arquiteturas supracitadas para um microprocessador hipotético de 8 bits². O objetivo é multiplicar dois números contidos nos endereços 0 e 3 da memória de um microprocessador hipotético e armazenar o resultado de volta na posição 0.

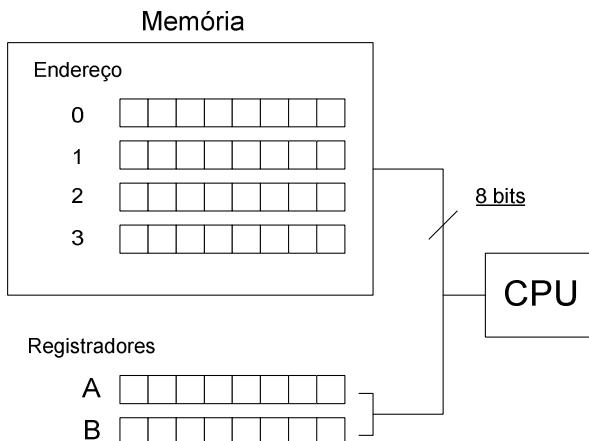


Fig. 1.4 – Diagrama esquemático para a comparação entre um microprocessador CISC e um RISC.

² Para um detalhamento completo ver: CARTER, Nicholas, *Computer Architecture*. 1 ed. New York: McGraw Hill, 2002.

Em microprocessadores CISC, o objetivo é executar a tarefa com o menor número de códigos possíveis (*assembly*). Assim, um microprocessador CISC hipotético poderia ter a seguinte instrução:

```
MULT 0,3      //multiplica o conteúdo do endereço 0 com o do endereço 3  
             //armazena o resultado no endereço 0.
```

Para um microprocessador RISC, a resolução do problema seria feita por algo como:

```
LOAD A,0      //carrega o registrador A com o conteúdo do endereço 0  
LOAD B,3      //carrega o registrador B com o conteúdo do endereço 3  
MULT A,B      //multiplica o conteúdo de A com o de B, o resultado fica em A  
STORE 0,A     //armazena o valor de A no endereço 0
```

O microprocessador RISC emprega o conceito de carga e armazenamento (*Load and Store*) utilizando registradores de uso geral. Os dados a serem multiplicados necessitam primeiro ser carregados nos registradores apropriados (A e B) e, posteriormente, o resultado é armazenado na posição de memória desejada (0). Em um microprocessador CISC, o dado pode ser armazenado sem a necessidade do uso explícito de registradores.

Esse conceito também pode ser visto quando se deseja escrever diretamente nos pinos de saída do microprocessador. Considerando que se deseja alterar os níveis lógicos (0 e 1) entre 8 pinos do microprocessador, expressos pela variável P1, o código resultante seria:

CISC:

```
MOV P1,0xAA    //escreve diretamente nos 8 pinos (P1) o valor binário 10101010
```

RISC:

```
LOAD A,0xAA    //carrega o registrador A com o valor binário 10101010  
OUT P1,A       //escreve o valor de A nos 8 pinos (P1)
```

Em um microprocessador RISC, antes de serem movidos para outro local, os valores sempre precisam passar pelos registradores de uso geral, por isso a caracterização da arquitetura como *Load and Store*.

Apesar dos microprocessadores RISC empregarem mais códigos para executar a mesma tarefa que os CISCs, ele são mais rápidos, pois geralmente executam uma instrução por ciclo de *clock*, ao contrário dos CISCs, que levam vários ciclos de *clock*. Todavia, o código em um microprocessador RISC será maior.

Os processadores RISC empregam mais transistores nos registradores de memória, enquanto que os CISC fazem uso de um grande número deles no seu conjunto de instruções.

O emprego de registradores nas operações de um processador RISC é uma vantagem, pois os dados podem ser utilizados posteriormente sem a necessidade de recarga, reduzindo o trabalho da CPU.

1.3 TIPOS DE MEMÓRIAS

Em sistemas digitais existem inúmeros tipos de memórias e, constantemente, novas são criadas para suprir a crescente demanda por armazenagem de informação. Basicamente, um sistema microprocessado contém duas memórias: a memória de programa, que armazenará o código a ser executado e a memória de dados, onde os dados de trabalho da CPU podem ser escritos e lidos.

Antigamente, a memória de programa era uma memória que só podia ser lida (ROM - *Read Only Memory*, memória somente de leitura) e que era gravada uma única vez (as memórias de gravação única se chamam OTP - *One Time Programmable*, programável somente uma vez). As memórias que eram regraváveis utilizavam raios ultravioleta para o seu apagamento e o chip possuía uma janela de quartzo para tal apagamento. Hoje em dia, se empregam memórias de programa apagáveis eletricamente, permitindo inúmeras gravações e regravações de forma rápida e sem a necessidade de equipamentos especiais.

Atualmente, nos microcontroladores são empregadas dois tipos de memórias regraváveis eletricamente: a memória *flash* EEPROM para o programa, e a *standart EEPROM* (*Electrical Erasable Programming Read Only Memory*) para armazenamento de dados que não devem ser perdidos. A diferença básica entre elas é que a memória *flash* só pode ser apagada por setores (vários bytes de uma única vez) e, na EEPROM, os bytes são apagados individualmente de forma mais lenta.

Para a armazenagem de dados que não precisam ficar retidos após a desenergização do circuito, se emprega uma memória volátil, chamada memória RAM (*Random Access Memory*, memória de acesso aleatório). Em microcontroladores, a memória usual é a SRAM (*Static RAM*), sendo fundamental para a programação, pois a maioria dos programas necessita de variáveis temporárias. Uma distinção importante é que a memória RAM não é a memória de dados da CPU, e sim uma memória para armazenagem temporária de dados do programa do usuário.

1.4 MICROCONTROLADORES

Um microcontrolador é o um sistema microprocessado com várias funcionalidades (periféricos) disponíveis em um único chip. Basicamente, um microcontrolador é um microprocessador com memórias de programa, de dados e RAM, temporizadores e circuitos de *clock* embutidos. O único componente externo que pode ser necessário é um cristal para determinar a frequência de trabalho. A grande vantagem de se colocar várias funcionalidades em um único circuito integrado é a possibilidade de desenvolvimento rápido de sistemas eletrônicos com o emprego de um pequeno número de componentes.

Dentre as funcionalidades encontradas nos microcontroladores, pode-se citar: gerador interno independente de *clock* (não necessita de cristal ou componentes externos); memória SRAM, EEPROM e *flash*; conversores analógicos-digitais (ADCs), conversores digitais-analógicos (DACs); vários

temporizadores/contadores; comparadores analógicos; saídas PWM; diferentes tipos de interface de comunicação, incluindo USB, USART, I2C, CAN, SPI, JTAG, Ethernet; relógio de tempo real; circuitos para gerenciamento de energia no chip; circuitos para o controle de inicialização (reset); alguns tipos de sensores; interface para LCD; e outros periféricos de acordo com o fabricante. Na fig. 1.5, é apresentado um diagrama esquemático de um microcontrolador típico.

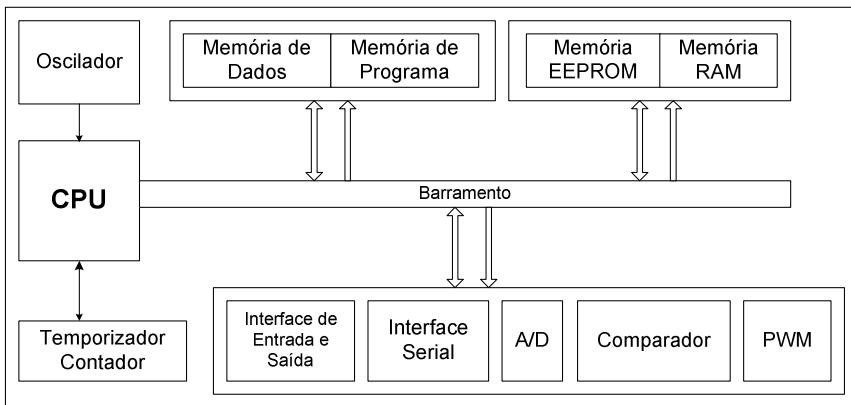


Fig. 1.5 – Diagrama esquemático de um microcontrolador típico.

Na tab. 1.1, é apresentada uma lista das várias famílias dos principais fabricantes de microcontroladores da atualidade. A coluna Núcleo indica o tipo de arquitetura ou unidade de processamento que constitui a base do microcontrolador. Dependendo do fabricante, existem ambientes de desenvolvimento que podem ser baixados gratuitamente da internet.

Tab. 1.1 – Principais fabricantes e famílias de microcontroladores (08/2011).

Fabricante	Sítio na Internet	Barram.	Família	Arquit.	Núcleo
Analog Device	www.analog.com	8 bits	ADuC8xx	CISC	8051
		32 bits	ADuC7xx	RISC	ARM7
		32 bits	ADuCRF101	RISC	Cortex M3
Atmel	www.atmel.com	8 bits	AT89xx	CISC	8051
		8 bits	AVR (ATtiny – ATmega – Xmega)	RISC	AVR8
		32 bits	AT32xx	RISC	AVR32
		32 bits	SAM7xx	RISC	ARM7
		32 bits	SAM9xx	RISC	ARM9
		32 bits	SAM3xx	RISC	Cortex M3
Cirrus Logic	www.cirrus.com	32 bits	EP73xx	RISC	ARM7
		32 bits	EP93xx	RISC	ARM9
Silicon Labs	www.silabs.com	8 bits	C8051xx	CISC	8051
Freescale	www.freescale.com	8 bits	RS08xx	CISC	
		8 bits	HCS08xx	CISC	6808
		8 bits	HC08xx - HC05 - HC11	CISC	
		16 bits	S12x - HC12 - HC16	CISC	
		32 bits	ColdFire	CISC	
		32 bits	68K	CISC	68000
		32 bits	K10-K70	RISC	Cortex M4
		32 bits	MPC5xxx	RISC	Power Arch
		32 bits	PXxxx	RISC	Power Arch
Fujitsu	www.fujitsu.com	8 bits	MB95xx	CISC	F ² MC-8FX
		16 bits	MB90xx	CISC	F ² MC-16FX
		32 bits	MB91xx	RISC	FR
		32 bits	FM3	RISC	Cortex M3
Holtek	www.holtek.com	8 bits	HT4x/5x/6x/8x/9x	RISC	
		8 bits	HT85F22x	CISC	8051
		32 bits	HT32F125x	RISC	Cortex M3
Infineon	www.infineon.com	8 bits	C500 - XC800	CISC	8051
		16 bits	C116 - XC166 - XC2x00	CISC	C116
		32 bits	TC11x	RISC	TriCore Arch
Maxim	www.maxim-ic.com	8 bits	DS 80/83/87/89xx	CISC	8051
		16 bits	MAXQxx	RISC	MAXQ20
		32 bits	ZA9Lx	RISC	ARM9
		32 bits	MAXQxx	RISC	MAXQ30
		32 bits	USIP	RISC	MIPS 4kSD
Microchip	www.microchip.com	8 bits	PIC 10/12/16/18	RISC	
		16 bits	PIC24 - dsPIC	RISC	
		32 bits	PIC32	RISC	MIPS32 M4K
National Semic.	www.national.com	8 bits	COP8	CISC	COP8
NXP	www.nxp.com	8 bits	80C51	CISC	8051
		16 bits	XA	CISC	
		32 bits	LPC 21/22/23/2400	RISC	ARM7
		32 bits	LPC 29/31/3200	RISC	ARM9
		32 bits	LPC 11/1200	RISC	Cortex M0
		32 bits	LPC 13/17/1800	RISC	Cortex M3
		32 bits	LPC 4300	RISC	Cortex M4

Fabricante	Sítio na Internet	Barram.	Família	Arquit.	Núcleo
Renesas	www.renesas.com	16 bits	RL78 / 78k /R8C/M16C	CISC	
		32 bits	M32C - H8S- RX	CISC	
		32 bits	V850 - SuperH	RISC	SH
ST	www.st.com	8 bits	STM8	RISC	STM8A
		8 bits	ST6/7	CISC	
		16 bits	ST10	CISC	
		32 bits	STR7	RISC	ARM7
		32 bits	STR9	RISC	ARM9
		32 bits	STM32F/W	RISC	Cortex M3
Texas Instruments	www.ti.com	16 bits	MSP430	RISC	
		32 bits	C2000	RISC	
		32 bits	Stellaris ARM	RISC	Cortex M3
		32 bits	TMS570	RISC	Cortex R4F
Toshiba	www.toshiba.com/taec	8 bits	TMP 86/87/88/89xx	CISC	
		16/32 bits	TMP 91/92/93/94/95xx	CISC	
		32 bits	TMPA9xx	RISC	ARM9
		32 bits	TMPM3xx	RISC	Cortex M3
Zilog	www.zilog.com	8 bits	Z8xx/Z8Encore/eZ80xx	CISC	Z80
		16 bits	ZNEO (Z16xx)	CISC	

Atualmente, nas arquiteturas modernas de **microcontroladores** há o domínio da **Harvard/RISC**, a qual evoluiu para a chamada arquitetura RISC avançada ou estendida. Ao contrário da RISC tradicional, essa é composta por um grande número de instruções, que utilizam uma quantidade reduzida de portas lógicas, produzindo um núcleo de processamento compacto, veloz e que permite uma programação eficiente (gera um menor número de linhas de código). Devido às questões de desempenho, compatibilidade eletromagnética e economia de energia, é importante que um microcontrolador execute a maioria das instruções em poucos ciclos de *clock*, diminuindo o consumo e a dissipação de energia.

1.5 OS MICROCONTROLADORES AVR

A história dos microcontroladores AVR começa em 1992 na Norwegian University of Science and Technology (NTNU), na Noruega, onde dois estudantes de doutorado, Alf-Egil-Boden e Vegard Wollan, defenderam uma tese sobre um microcontrolador de 8 bits com memória de programa *flash* e arquitetura RISC avançada. Dos seus nomes surgiu a sigla AVR: Alf – Vegard – RISC. Percebendo as vantagens do microcontrolador que haviam

criado, Alf e Vegard refinaram o seu projeto por dois anos. Com o AVR aperfeiçoado, foram para a Califórnia (EUA) para encontrar um patrocinador para a ideia, a qual foi comprada pela Atmel, que lançou comercialmente o primeiro AVR em meados de 1997, o AT90S1200. Desde então, o aperfeiçoamento do AVR é crescente, juntamente com a sua popularidade.

O AVR apresenta uma boa eficiência de processamento e um núcleo compacto (poucos milhares de portas lógicas). O desempenho do seu núcleo de 8 bits é superior ao da maioria das outras tecnologias de 8 bits disponíveis atualmente no mercado³. Com uma arquitetura RISC avançada, o AVR apresenta mais de uma centena de instruções e uma estrutura voltada à programação C, permitindo a produção de códigos compactos (menor número de bytes por funcionalidade). Também, apresenta inúmeros periféricos que o tornam adequado para uma infinidade de aplicações.

As principais características dos microcontroladores AVR são:

- executam a maioria das instruções em 1 ou 2 ciclos de *clock* (poucas em 3 ou 4) e operam com tensões entre 1,8 V e 5,5 V, com velocidades de até 20 MHz. Estão disponíveis em diversos encapsulamentos;
- alta integração e grande número de periféricos com efetiva compatibilidade entre toda a família AVR;
- possuem vários modos para redução do consumo de energia e características adicionais (*picoPower*) para sistemas críticos;
- possuem 32 registradores de propósito geral e instruções de 16 bits (cada instrução ocupa 2 bytes na memória de programa);
- memória de programação *flash* programável *in-system*, SRAM e EEPROM;
- facilmente programados e com *debug in-system* via interface simples, ou com interfaces JTAG compatível com 6 ou 10 pinos;
- um conjunto completo e gratuito de softwares;
- preço acessível.

³ Recentemente a ST Microelectronics lançou o STM8, um forte concorrente, em termos de desempenho, do AVR de 8 bits.

Existem microcontroladores AVR específicos para diversas áreas, tais como: automotiva, controle de LCDs, redes de trabalho CAN, USB, controle de motores, controle de lâmpadas, monitoração de bateria, 802.15.4/ZigBee™ (comunicação sem fio, *wireless*) e controle por acesso remoto.

1.6 A FAMÍLIA AVR

Dentre os principais componentes da família AVR, pode-se citar:

- **tinyAVR® - ATtiny** – 4 até 28 pinos de I/O.
Microcontroladores de propósito geral de até 8 kbytes de memória *flash*, 512 bytes de SRAM e EEPROM.
- **megaAVR® - ATmega** – 23 até 86 pinos de I/O.
Microcontroladores com vários periféricos, multiplicador por hardware, até 256 kbytes de memória *flash*, 4 kbytes de EEPROM e 8 kbytes de SRAM.
- **picoPower™ AVR.** Microcontroladores com características especiais para economia de energia (são designados com a letra P, como por exemplo, ATmega328P).
- **XMEGA™ ATxmega** – 50 até 78 pinos de I/O.
Os microcontroladores XMEGA dispõem de avançados periféricos para o aumento de desempenho, tais como: DMA (*Direct Memory Access*) e sistema de eventos.
- **AVR32** (não pertence às famílias acima) – 28 até 160 pinos de I/O.
Microcontroladores de 32 bits com arquitetura RISC projetada para maior processamento por ciclos de *clock*, com eficiência de 1,3 mW/MHz e até 210 DMIPS (Dhrystone Million Instructions per Second⁴) a 150 MHz, conjunto de instruções para DSP (*Digital Signal Processing*, processamento digital de sinais) com SIMD (*Single Instruction Multiple Data*, instrução simples dados múltiplos), soluções SoC (*System-on-a-chip*, sistema em um chip) e suporte completo ao Linux.

As principais características para alguns dos AVRs ATtiny e ATmega são apresentadas nas tabs. 1.2 e 1.3. Para aplicações que exijam outras funcionalidades de hardware além das apresentadas, o sítio do fabricante deve ser consultado.

⁴ **Dystone** é um tipo de programa, chamado *benchmark*, desenvolvido para testar o desempenho de processadores. O DMIPS é obtido quando a pontuação obtida do Dystone é dividida por 1,757 (o número de Dystone por segundo obtida de uma máquina nominal de 1 MIPS – 1 Milhão de Instruções por Segundo).

Tab. 1.2 – Comparação entre alguns ATtiny^{*}.

ATtiny	Flash (Kbytes)	EEPROM (Bytes)	SRAM (Bytes)	Max Pinos I/O	Canais AD 10-bits	16-bit TC	8-bit TC	Canais PWM	SPI	TWI	F.max (MHz)	VCC (V)
ATtiny10	1	--	32	4	--	1	--	2	--	--	12	1,8 - 5,5
ATtiny12	1	64	--	6	--	--	1	--	--	--	8	1,8 - 5,5
ATtiny13A	1	64	64	6	4	--	1	2	--	--	20	1,8 - 5,5
ATtiny2313	2	128	128	18	--	1	1	4	USI	USI	20	1,8 - 5,5
ATtiny24	2	128	128	12	8	1	1	4	USI	USI	20	1,8 - 5,5
ATtiny24A	2	128	128	12	8	1	1	4	USI	USI	20	1,8 - 5,5
ATtiny25	2	128	128	6	4	--	2	4	USI	USI	20	1,8 - 5,5
ATtiny261A	2	128	128	16	11	1	2	2	sim	USI	20	1,8 - 5,5
ATtiny28L	2	--	32	11	--	--	1	--	--	--	4	1,8 - 5,5
ATtiny4	0,5	--	32	4	--	1	--	2	--	--	12	1,8 - 5,5
ATtiny43U	4	64	256	16	4	--	2	4	sim	--	8	0,7 - 1,8
ATtiny44A	4	256	256	12	8	1	1	4	USI	USI	20	1,8 - 5,5
ATtiny45	4	256	256	6	4	--	2	4	USI	USI	20	1,8 - 5,5
ATtiny461A	4	256	256	16	11	1	2	2	sim	USI	20	1,8 - 5,5
ATtiny48	4	64	256	24/28	6/8	1	1	1	sim	sim	12	1,8 - 5,5
ATtiny5	0,5	--	32	4	--	1	--	2	--	--	12	1,8 - 5,5
ATtiny84	8	512	512	12	8	1	1	4	USI	USI	20	1,8 - 5,5
ATtiny85	8	512	512	6	4	--	2	4	USI	USI	20	1,8 - 5,5
ATtiny861A	8	512	512	16	11	1	2	2	sim	USI	20	1,8 - 5,5
ATtiny88	8	64	512	24/28	6/8	1	1	1	sim	sim	12	1,8 - 5,5
ATtiny9	1	--	32	4	--	1	--	2	--	--	12	1,8 - 5,5

*Os ATtiny possuem oscilador interno RC e comparador analógico.

Tab. 1.3 – Comparaçāo entre alguns ATmega*.

ATmega	Flash (Kbytes)	EEPROM (Kbytes)	SRAM (Bytes)	Max Pinos	Canais AD	16bit TC	8-bit TC	Canais PWM	SPI	TWI	USART	F. _{max} (MHz)	VCC (V)
Atmega1280	128	4	8192	86	16	4	2	16	1+USART	sim	4	16	1,8 - 5,5
Atmega1281	128	4	8192	54	8	4	2	9	1+USART	sim	2	16	1,8 - 5,5
Atmega1284P	128	4	16K	32	8	2	2	6	sim	sim	2	20	1,8 - 5,5
Atmega128A	128	4	4096	53	8	2	2	8	1	sim	2	16	2,7 - 5,5
Atmega162	16	0,5	1024	35	--	2	2	6	1	--	2	16	1,8 - 5,5
Atmega164P	16	0,5	1024	32	8	1	2	6	1+USART	sim	2	20	1,8 - 5,5
Atmega165P	16	0,5	1024	54	8	1	2	4	1+USI	USI	1	16	1,8 - 5,5
Atmega168PA	16	0,5	1024	23	8	1	2	6	1+USART	sim	1	20	1,8 - 5,5
Atmega16A	16	0,5	1024	32	8	1	2	4	1	sim	1	16	2,7 - 5,5
Atmega2560	256	4	8192	86	16	4	2	16	1+USART	sim	4	16	1,8 - 5,5
Atmega2561	256	4	8192	54	8	4	2	9	1+USART	sim	2	16	1,8 - 5,5
Atmega324P	32	1	2048	32	8	1	2	6	1+USART	sim	2	20	1,8 - 5,5
Atmega3250	32	1	2048	69	8	1	2	4	1+USI	USI	1	16	1,8 - 5,5
Atmega3250P	32	1	2048	69	8	1	2	4	1+USI	USI	1	20	1,8 - 5,5
Atmega325P	32	1	2048	54	8	1	2	4	1+USI	USI	1	20	1,8 - 5,5
Atmega328P	32	1	2048	23	8	1	2	6	1+USART	sim	1	20	1,8 - 5,5
Atmega32A	32	1	2048	32	8	1	2	4	1	sim	1	16	2,7 - 5,5
Atmega48PA	4	0,25	512	23	8	1	2	6	1+USART	sim	1	20	1,8 - 5,5
Atmega64	64	2	4096	54	8	2	2	8	1	sim	2	16	2,7 - 5,5
Atmega640	64	4	8192	86	16	4	2	16	1+USART	sim	4	16	1,8 - 5,5
Atmega644	64	2	4096	32	8	1	2	6	1+USART	sim	1	20	1,8 - 5,5
Atmega645	64	2	4096	54	8	1	2	4	1+USI	USI	1	16	1,8 - 5,5
Atmega6450	64	2	4096	69	8	1	2	4	1+USI	USI	1	16	1,8 - 5,5
Atmega64A	64	2	4096	54	8	2	2	8	1	sim	2	16	2,7 - 5,5
Atmega8515	8	0,5	512	35	--	1	1	3	1	--	1	16	2,7 - 5,5
Atmega8535	8	0,5	512	32	8	1	2	4	1	sim	1	16	2,7 - 5,5
Atmega88PA	8	0,5	1024	23	8	1	2	6	1+USART	sim	1	20	1,8 - 5,5
Atmega8A	8	0,5	1024	23	6/8	1	2	3	1	sim	1	16	2,7 - 5,5

* Os ATmega possuem oscilador interno RC, comparador analógico, RTc (*Real Time counter*) e multiplicador por hardware.

2. O ATMEGA328

Neste capítulo, apresentam-se, sucintamente, o microcontrolador ATmega328, suas funcionalidades e a descrição de seu hardware: memórias, pinos, sistema de *clock*, sistema de *reset* e sistema de gerenciamento de energia.

O ATmega328⁵ será abordado por ser um microcontrolador compacto que apresenta a maioria das características da família AVR e uma memória *flash* de maior tamanho comparado aos microcontroladores AVR com o mesmo número de pinos. Outro fator decisivo, é que ele é utilizado atualmente na plataforma de desenvolvimento Arduino, de fácil aquisição. O importante a saber é que, ao se programar esse microcontrolador, os conceitos de programação para qualquer outro da família AVR serão iguais, devido às similaridades entre os componentes. Na maioria dos casos a mudança se dá no uso e nome de alguns registradores e periféricos. Na dúvida, qualquer mudança de hardware e software pode ser resolvida com uma busca ao manual do fabricante.

As principais características do ATmega328 são:

- Microcontrolador de baixa potência, com arquitetura RISC avançada.
- 131 instruções, a maior parte executada em 1 ou 2 ciclos de *clock* (poucas em 3 ou 4 ciclos).
- 32 registradores de trabalho de propósito geral (8 bits cada). Alguns trabalham em par para endereçamentos de 16 bits.
- Operação de até 20 MIPS a 20 MHz.
- Multiplicação por hardware em 2 ciclos de *clock*.
- 32 kbytes de memória de programa *flash* de auto programação *In-System* (8 k, 16 k, nos respectivos ATmega88 e ATmega168).
- 1 kbytes de memória EEPROM.
- 2 kbytes de memória SRAM.

⁵ Os ATmega48/88/168/328 são idênticos com diferenças sutis, principalmente na quantidade de memória disponível. A Atmel disponibiliza um único manual para eles.

- Ciclos de escrita e apagamento: memória *flash* 10 mil vezes, EEPROM 100 mil vezes.
- Seção opcional para código de *boot* para programação *In-System* por *boot loader*⁶.
- Bits de bloqueio para proteção contra a cópia do firmware.
- Possui os seguintes periféricos:
 - ➔ 23 entradas e saídas (I/Os) programáveis.
 - ➔ 2 Temporizadores/Contadores de 8 bits com *Prescaler* separado, com modo de comparação.
 - ➔ 1 Temporizador/Contador de 16 bits com *Prescaler* separado, com modo de comparação e captura.
 - ➔ Contador de tempo real (com um cristal externo de 32,768 kHz conta precisamente 1 s).
 - ➔ 6 canais PWM.
 - ➔ 8 canais AD com resolução de 10 bits na versão TQFP (*Thin profile plastic Quad Flat Package*) e 6 canais na versão PDIP (*Plastic Dual Inline Package*).
 - ➔ Interface serial para dois fios orientada a byte (TWI), compatível com o protocolo I2C.
 - ➔ Interface serial USART.
 - ➔ Interface serial SPI *Master/Slave*.
 - ➔ *Watchdog Timer* com oscilador interno separado.
 - ➔ 1 comparador analógico.
- Características especiais:
 - ➔ *Power-on Reset* e detecção *Brown-out* programável.
 - ➔ Oscilador interno RC (não há a necessidade do uso de cristal externo ou de outra fonte de *clock*).
 - ➔ Fontes de interrupções internas e externas (em todos os pinos de I/O).
 - ➔ Saída de *clock* em um pino de I/O (PB0).
 - ➔ *Pull-up* habilitáveis em todos os pinos de I/O
 - ➔ 6 modos de *Sleep*: *Idle*, Redução de ruído do ADC, *Power-down*, *Power-save*, *Standby* e *Extended Standby*.
 - ➔ Medição de temperatura do encapsulamento.
- Tensão de operação: 1,8 - 5,5 V.
- Consumo de corrente a 1 MHz (1,8 V, 25 °C): modo ativo = 0,2 mA e modo *Power-down* = 0,1 µA.

⁶ **Boot loader** é um pequeno programa que pode ser escrito no início ou no final da memória de programa e serve para que o microcontrolador gerencie a gravação de sua memória. Para tal, é necessário uma interface de comunicação externa com o software de desenvolvimento. No AVR o espaço destinado ao *boot loader* fica no final da memória de programa e a comunicação se dá através de um dos seus periféricos de comunicação, a USART.

Na fig. 2.1, é apresentado o diagrama esquemático da estrutura interna de um ATmega328. O barramento de dados é de 8 bits, caracterizando o número de bits do microcontrolador. As instruções do ATmega são de 16 ou 32 bits (a maioria é de 16 bits). Assim, cada instrução consome 2 ou 4 bytes na memória de programa (um byte par e um ímpar). O acesso às posições de memória, dado pelo contador de programa (*Program Counter - PC*), é realizada de dois em dois bytes, começando sempre por uma posição par. Portanto, o barramento de endereços deve ser capaz de endereçar sempre posições pares da memória de programa. Logo, o bit menos significativo do barramento de endereços pode ser desprezado. Desta forma, para a memória de 32 kbytes do Atmega328 são necessários 14 bits de endereçamento ($2^{14} = 16.384$ endereços) e não 15 ($2^{15} = 32.768$ endereços). Na fig. 2.1, é ilustrada a arquitetura Harvard empregada pelo ATmega328 (barramento de instruções e dados separados).

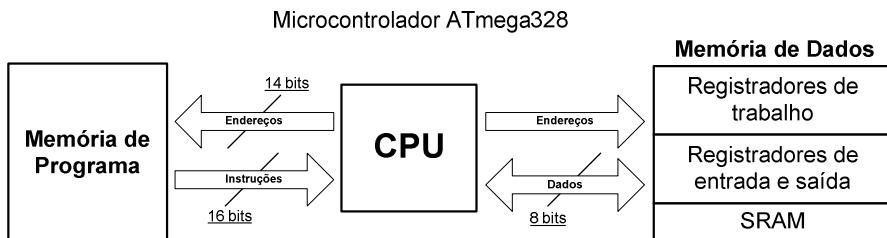


Fig. 2.1 – Diagrama esquemático da estrutura de um microcontrolador ATmega328.

O núcleo AVR utiliza 32 registradores de trabalho diretamente conectados à Unidade Lógico-Aritmética (ALU – *Arithmetic Logic Unit*), permitindo que dois registradores independentes sejam acessados com uma simples instrução em um único ciclo de *clock*. Seis desses registradores podem ser usados como registradores de endereçamento indireto de 16 bits (ponteiros para o acesso de dados), denominados X, Y e Z. Na fig. 2.2, são ilustrados esses registradores e seus respectivos endereços na memória de dados. Os 32 registradores não podem ser

empregados em todas as instruções do microcontrolador, pois algumas instruções empregam registradores específicos para o seu trabalho (ver o apêndice A).

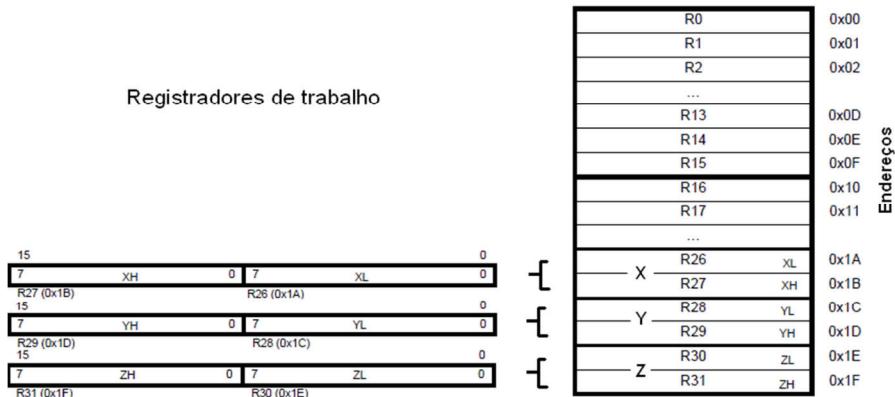


Fig. 2.2 – Registradores de trabalho da CPU do ATmega.

A função principal da Unidade Central de Processamento (CPU) é garantir a correta execução do programa, sendo capaz de acessar as memórias, executar cálculos, controlar os periféricos e tratar interrupções.

A cada pulso de *clock* o contador do programa incrementa em uma unidade, cujo valor corresponde a um endereço da memória *flash* de programa. A instrução correspondente é, então, lida, decodificada e executada pela CPU. Todo o trabalho é ditado pelo sinal de *clock*; quando ocorre algum desvio no programa o valor do contador de endereços é alterado e a CPU executa a instrução no novo endereço.

O diagrama em blocos da CPU do AVR, incluindo os periféricos, pode ser visto na fig. 2.3.

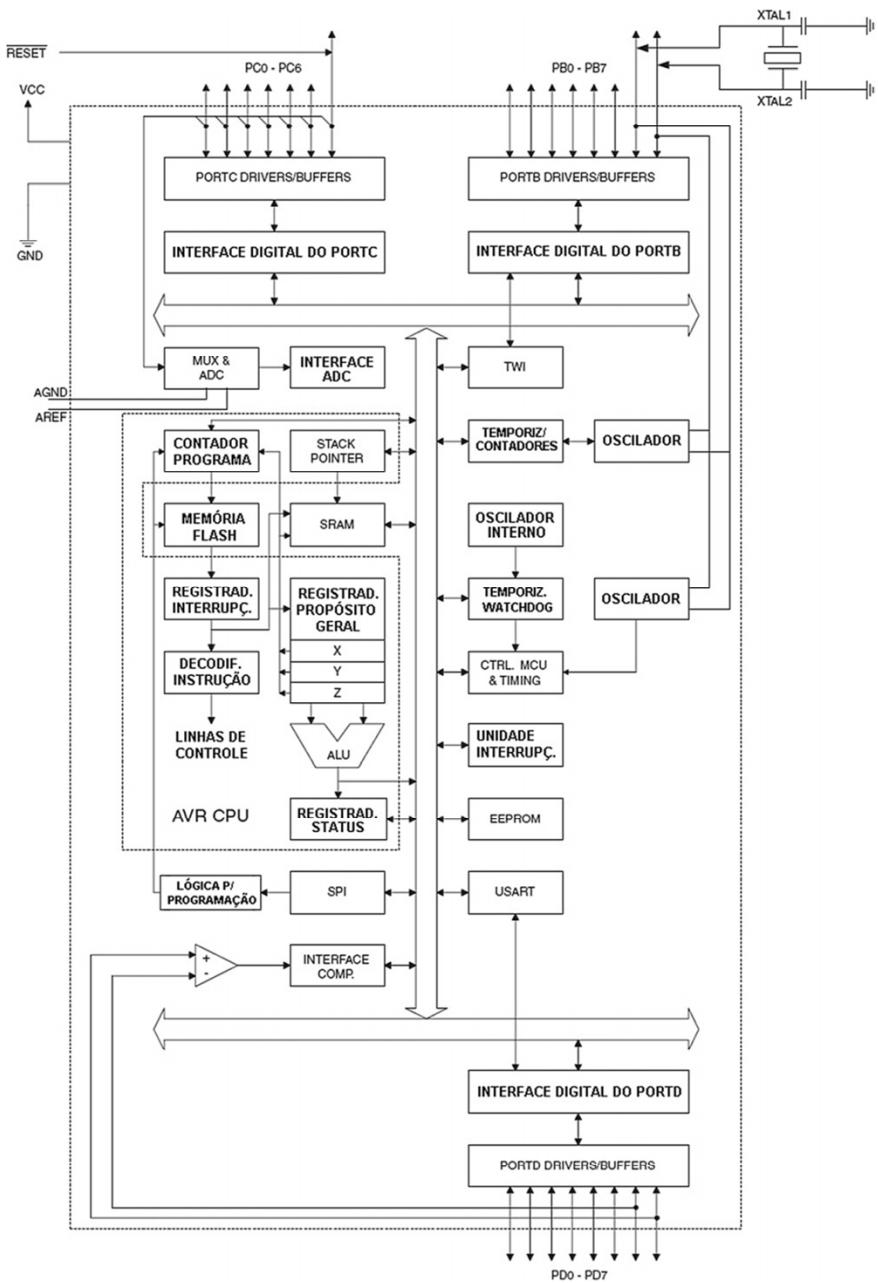


Fig. 2.3 – Diagrama em blocos do ATmega328.

A arquitetura do AVR permite a busca e execução de instruções em paralelo devido ao emprego da técnica de *Pipeline*. Assim, o desempenho alcançado pode chegar a 1 MIPS por MHz. O diagrama de tempo da busca e execução de instruções, com referência ao *clock* da CPU, é apresentado na fig. 2.4. Observa-se que uma instrução é executada enquanto a próxima é lida. O ATmega emprega um *pipeline* de 2 estágios, dessa forma uma instrução é lida e decodificada dentro de um mesmo ciclo de *clock*, enquanto a instrução anterior é executada.

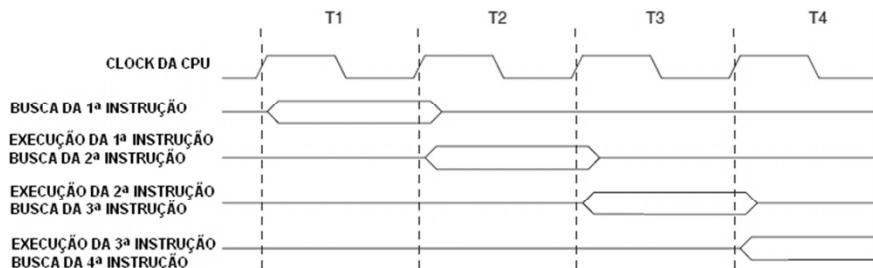


Fig. 2.4 – Diagrama de tempo para a busca e execução de instruções no ATmega.

2.1 AS MEMÓRIAS

Um diagrama da memória de dados e memória SRAM do ATmega328 pode ser visto na fig. 2.5. A organização da memória é linear, começando no endereço 0 e indo até o endereço 2303 (0x8FF). Destas posições de memória, 32 pertencem aos registradores de uso geral (0x000 até 0x01F, fig. 2.2), 64 aos registradores de entrada e saída (0x020 até 0x05F) e 2048 bytes pertencem à memória SRAM (0x060 até 0x8FF), cujos 160 primeiros endereços são empregados para a extensão dos registradores de entrada e saída. Isso é necessário porque o número de periféricos no ATmega328 é superior ao que pode ser suportado pelos 64 registradores originais (dos primeiros ATmegas) e, também, para permitir o acréscimo de futuras

funcionalidades. Desse modo, os engenheiros da Atmel utilizaram a SRAM⁷ para aumentar o número de registradores de I/O sem a necessidade de alterações profundas no projeto do chip.

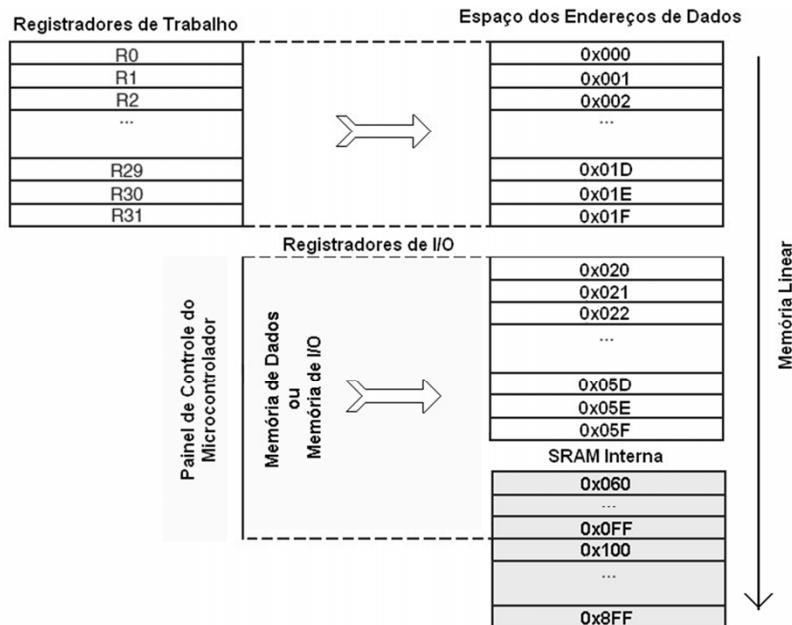


Fig. 2.5 – Memória de dados e memória SRAM do ATmega328.

Os registradores de I/O são o painel de controle do microcontrolador, pois todas as configurações de trabalho, incluindo acesso às entradas e saídas, se encontram nessa parte da memória. É fundamental compreender que são esses registradores que dão acesso às funcionalidades do microcontrolador, pois possuem todas as informações referentes aos periféricos e ao processamento da CPU. É com esses registradores que o programador terá que se familiarizar para trabalhar com os periféricos (as ‘chaves’ que ligam e desligam tudo). Os registradores

⁷ Literalmente, a memória SRAM não pode ser denominada memória de dados ou memória de I/O da CPU, pois ela é projetada para armazenar variáveis temporárias do programa e não os registradores de I/O que possuem endereço fixo.

são apresentados na tab. 2.1 (observar seu endereçamento na fig. 2.5) e, dado sua importância, os mesmos serão vistos com detalhes posteriormente (um resumo é incluído no apêndice G).

Tab. 2.1: Registradores de entrada e saída da memória de dados (painel de controle do microcontrolador)*.

End.	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x23	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
0x24	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x25	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x26	PINC	-	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
0x27	DDRC	-	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
0x28	PORTC	-	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
0x29	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
0x2A	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
0x2B	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
0x35	TIFR0	-	-	-	-	-	OCF0B	OCF0A	TOV0
0x36	TIFR1	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1
0x37	TIFR2	-	-	-	-	-	OCF2B	OCF2A	TOV2
0x3B	PCIFR	-	-	-	-	-	PCIF2	PCIF1	PCIF0
0x3C	EIFR	-	-	-	-	-	-	INTF1	INTF0
0x3D	EIMSK	-	-	-	-	-	-	INT1	INT0
0x3E	GPIOR0	Registrador de I/O de propósito geral 0							
0x3F	EECR	-	-	EEP1	EEP0	EERIE	EEMPE	EEPE	EERE
0x40	EEDR	Registrador de dados da EEPROM							
0x41	EEARL	Byte menor do registrador de endereço da EEPROM							
0x42	EEARH	Byte maior do registrador de endereço da EEPROM							
0x43	GTCCR	TSM	-	-	-	-	-	PSRASY	PSRSYNC
0x44	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
0x45	TCCR0B	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00
0x46	TCNT0	Temporizador/Contador 0 (8 bits) – Registrador de contagem							
0x47	OCR0A	Registrador de comparação de saída A do Temporizador/Contador 0							
0x48	OCR0B	Registrador de comparação de saída B do Temporizador/Contador 0							
0x4A	GPIOR1	Registrador de I/O de propósito geral 1							
0x4B	GPIOR2	Registrador de I/O de propósito geral 2							
0x4C	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
0x4D	SPSR	SPIF	WCOL	-	-	-	-	-	SPI2X
0x4E	SPDR	Registrador de dados da SPI							
0x50	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACISO
0x53	SMCR	-	-	-	-	SM2	SM1	SM0	SE
0x54	MCUSR	-	-	-	-	WDRF	BORF	EXTRF	PORF
0x55	MCUCR	-	BODS	BODSE	PUD	-	-	IVSEL	IVCE
0x57	SPMCSR	SPMIE	RWWSB	-	RWWRE	BLBSET	PGWRT	PGERS	SELFPRGEN
0x5D	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
0x5E	SPH	-	-	-	-	-	SP10	SP9	SP8
0x5F	SREG	I	T	H	S	V	N	Z	C
0x60	WDTCSR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDPO
0x61	CLKPR	CLKPCE	-	-	-	CLKPS3	CLKPS2	CLKPS1	CLKPS0
0x64	PRR	PRTWI	PRTIM2	PRTIM0	-	PRTIM1	PRSPI	PRUSART0	PRADC
0x66	OSCCAL	Registrador de calibração do oscilador							
0x68	PCICR	-	-	-	-	-	PCIE2	PCIE1	PCIE0
0x69	EICRA	-	-	-	-	ISC11	ISC10	ISC01	ISC00
0x6B	PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
0x6C	PCMSK1	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
0x6D	PCMSK2	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
0x6E	TIMSK0	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0

End.	Nome	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x6F	TIMSK1	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1
0x70	TIMSK2	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2
0x78	ADCL	Registrador de dados do ADC, byte menor							
0x79	ADCH	Registrador de dados do ADC, byte maior							
0x7A	ADCSSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
0x7B	ADCSSRB	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0
0x7C	ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0
0x7E	DIDR0	-	-	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D
0x7F	DIDR1	-	-	-	-	-	-	AIN1D	AIN0D
0x80	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10
0x81	TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
0x82	TCCR1C	FOC1A	FOC1B	-	-	-	-	-	-
0x84	TCNT1L	Temporizador/Contador 1 (16 bits) – Registrador de contagem do byte menor							
0x85	TCNT1H	Temporizador/Contador 1 (16 bits) – Registrador de contagem do byte maior							
0x86	ICR1L	Temporizador/Contador 1 (16 bits) – Registrador de captura de entrada, byte menor							
0x87	ICR1H	Temporizador/Contador 1 (16 bits) – Registrador de captura de entrada, byte maior							
0x88	OCR1AL	Temporizador/Contador 1 (16 bits) – Registrador da saída de comparação A, byte menor							
0x89	OCR1AH	Temporizador/Contador 1 (16 bits) – Registrador da saída de comparação A, byte maior							
0x8A	OCR1BL	Temporizador/Contador 1 (16 bits) – Registrador da saída de comparação B, byte menor							
0x8B	OCR1BH	Temporizador/Contador 1 (16 bits) – Registrador da saída de comparação B, byte maior							
0x80	TCCR2A	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20
0x81	TCCR2B	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20
0xB2	TCNT2	Temporizador/Contador 2 (8 bits) – Registrador de contagem							
0xB3	OCR2A	Temporizador/Contador 2 (8 bits) – Registrador da saída de comparação A							
0xB4	OCR2B	Temporizador/Contador 2 (8 bits) – Registrador da saída de comparação B							
0xB6	ASSR	-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB
0xB8	TWBR	Registrador da taxa de dados da interface serial a dois fios - TWI							
0xB9	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0
0xBA	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
0xBB	TWDR	Registrador de dados da interface serial a dois fios - TWI							
0xBC	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
0xBD	TWAMR	TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	-
0xC0	UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
0xC1	UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
0xC2	UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USB0	UCSZ01 / UDORD0	UCSZ00 / UCPhAO	UCPOL0
0xC4	UBRR0L	Registrador da taxa de transmissão da USART, byte menor							
0xC5	UBRR0H	-	-	-	-	Registrador da taxa de transmissão da USART, byte maior			
0xC6	UDR0	Registrador I/O de dados da USART							

* Os endereços que não aparecem são reservados e foram suprimidos para não ocupar espaço na tabela, em negrito os que os seguem.

A organização da memória de programa pode ser vista na fig. 2.6. Cada endereço da memória possui 2 bytes, pois as instruções do AVR são de 16 ou 32 bits. Dessa forma, a memória possui um total de 16384 endereços (de 0x0000 até 0x3FFF), correspondendo a 32 kbytes de memória. Existe uma seção específica para carregar o *boot loader*, que pode ou não ser utilizada para esse fim. A memória flash suporta, no mínimo, 10 mil ciclos de escrita e apagamento.

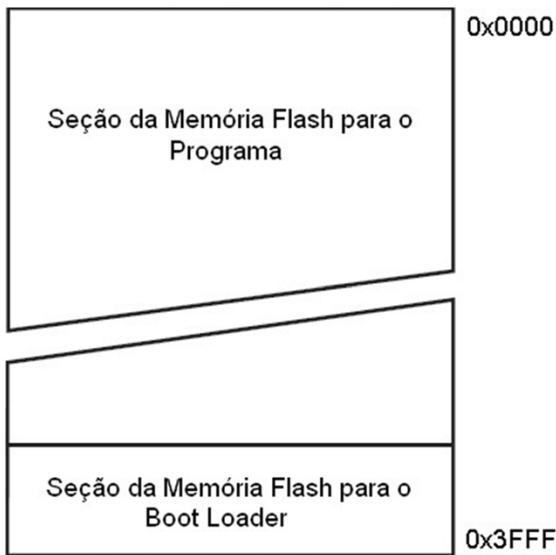


Fig. 2.6 – Organização da memória de programa.

A memória EEPROM é de 1 kbytes e é organizada separadamente. Cada byte individual pode ser lido ou escrito e a memória suporta, no mínimo, 100 mil ciclos de escrita e apagamento.

2.1.1 O REGISTRADOR DE STATUS - SREG

Um dos registradores mais importantes do painel de controle do microcontrolador é o SREG (Status Register), que indica, através de bits individuais, o estado das operações lógicas e aritméticas da CPU e permite habilitar ou não as interrupções (chave geral). Seus bits são empregados na programação para tomadas de decisões e na realização de operações lógico-aritméticas. O detalhamento do registrador SREG é apresentado a seguir.

Bit	7	6	5	4	3	2	1	0
SREG	I	T	H	S	V	N	Z	C
Lê/Escrive	L/E							
Valor Inicial	0	0	0	0	0	0	0	0

Bit 7 – I – Global Interrupt Enable

Esse bit é a chave geral para habilitar as interrupções. Cada interrupção individual possui seus registradores de controle. O bit **I** é limpo quando uma interrupção ocorre (impedindo que outras ocorram simultaneamente) e volta a ser ativo quando se termina o tratamento da interrupção (instrução RETI).

Bit 6 – T – Bit Copy Storage

Serve para copiar o valor de um bit de um registrador ou escrever o valor de um bit em um registrador (instruções BLD e BST).

Bit 5 – H – Half Carry Flag

Indica quando um *Carry* auxiliar (em um *nibble*⁸) ocorreu em alguma operação aritmética. Útil em aritmética BCD (Binário Codificado em Decimal).

Bit 4 – S – Sign Bit, $S = N \oplus V$

O bit S é o resultado de uma operação ou-exclusivo entre o bit de sinalização negativo **N** e o bit de sinalização de estouro do complemento de dois **V**.

Bit 3 – V – Two's Complement Overflow Flag

O bit de sinalização de estouro do complemento de dois, ajuda na aritmética em complemento de dois.

Bit2 – N – Negative Flag

O bit de sinalização negativo indica quando uma operação aritmética ou lógica resulta em um valor negativo.

Bit1 – Z – Zero Flag

O bit de sinalização zero indica quando uma operação aritmética ou lógica resulta em zero.

Bit 0 – C – Carry Flag

O bit de sinalização de *Carry* indica quando houve um estouro numa operação aritmética.

⁸ Um *nibble* = 4 bits, dois *nibbles* = 1 byte.

2.1.2 O STACK POINTER

O *Stack Pointer* (SP, ponteiro de pilha) é um registrador que armazena um endereço correspondente a uma posição da memória RAM, a qual é utilizada na forma de uma pilha para armazenagem temporária de dados: variáveis locais e endereços de retorno após chamadas de sub-rotinas e interrupções. Em resumo, o SP indica a posição onde um determinado dado foi armazenado na pilha alocada na RAM, esse conceito é ilustrado na fig. 2.7.

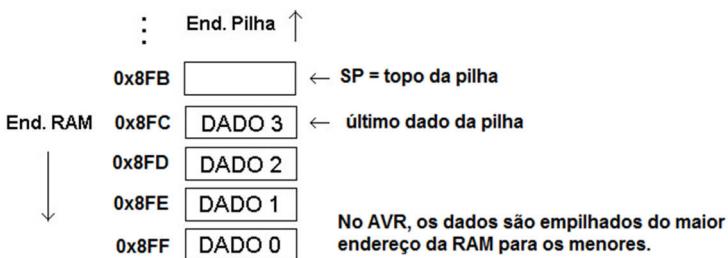


Fig. 2.7 – *Stack Pointer*.

O endereço do SP é pós-decrementado toda vez que um dado é colocado na pilha. Assim, a cada novo dado colocado na pilha, o endereço do SP apresenta um valor menor que o anterior. Da mesma forma, quando um dado é retirado da pilha, o endereço do SP é pré-incrementado, sempre apontando uma posição acima do último dado válido da pilha. Isso implica que o comando PUSH, que coloca um dado na pilha, diminui o valor do SP e o comando POP, que retira um dado da pilha, o incrementa. Na tab. 2.2, são apresentadas as instruções do ATmega que afetam o SP.

Dado o funcionamento do SP, na sua inicialização ele deve apontar para o endereço final da RAM, no caso do ATmega328 o endereço é 0x8FF (seu valor default após a energização). Entretanto, existem microcontroladores da família ATmega que precisam ter o SP inicializado pelo programador. Nesse caso, quando o programa é escrito em *assembly* a

inicialização deve ser feita pelo programador de forma explícita (ver os exemplos de programação *assembly* no capítulo 5). Essa inicialização é feita automaticamente quando o programa é escrito na linguagem C, pois o compilador se encarrega da tarefa.

Tab. 2.2: Instruções do ATmega que afetam o *Stack Pointer*.

Instrução	<i>Stack Pointer</i>	Descrição
PUSH	Decrementa 1	Um dado é colocado na pilha (1 byte).
CALL ICALL RCALL	Decrementa 2	O endereço de retorno é colocado na pilha quando uma chamada de sub-rotina ou interrupção acontece (o endereço possui 2 bytes).
POP	Incrementa 1	O dado do topo da pilha é retirado (1 byte).
RET RETI	Incrementa 2	O endereço de retorno é retirado da pilha quando se retorna de uma sub-rotina ou interrupção (o endereço possui 2 bytes).

Como o SP armazena um endereço da RAM, ele deve ter um número de bits suficiente para isso. Como o ATmega possui registradores de 8 bits, são necessários dois registradores para o SP, um armazena a parte baixa do endereço (*SP Low*) e outro armazena a parte alta do endereço (*SP High*), resultando num registrador de 12 bits (os 4 bits mais significativos do SPH não são utilizados). Abaixo são apresentados os registradores do SP (ver os endereços 0x5D e 0x5E da tab. 2.1).

Bit	15	14	13	12	11	10	9	8
SPH	-	-	-	-	SP11	SP10	SP9	SP8
SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
Bit	7	6	5	4	3	2	1	0
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	-	-	-	-	1	0	0	0
	1	1	1	1	1	1	1	1

Fig. 2.8- Detalhamento dos registradores do *Stack Pointer* (valor inicial 0x8FF).

2.2 DESCRIÇÃO DOS PINOS

Na fig. 2.9, os nomes dos pinos do ATmega328 são apresentados para os encapsulamentos PDIP e TQFP. Cada pino acumula várias funções selecionáveis, as siglas nos pinos resumem as funcionalidades desses e serão abordadas no momento adequado. A tab. 2.3 contém a descrição sucinta dos referidos pinos.

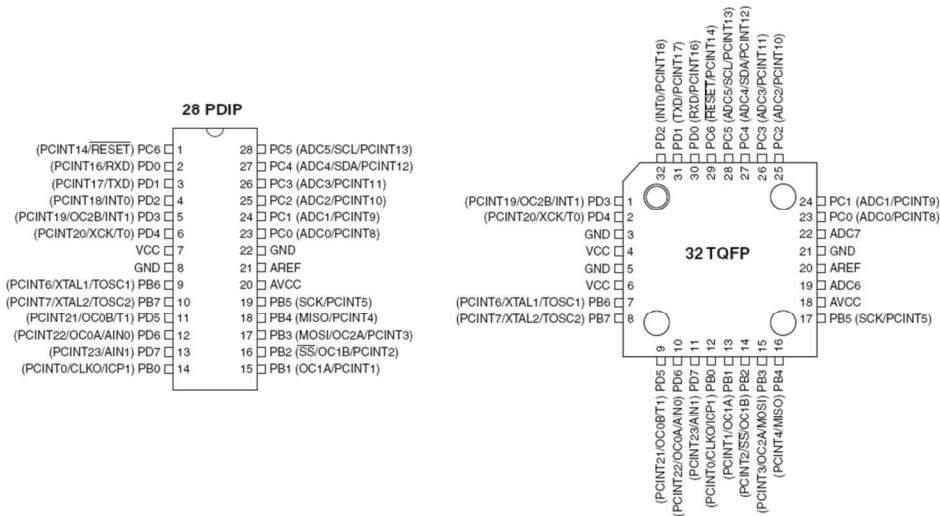


Fig. 2.9- Encapsulamentos PDIP e TQFP para o ATmega328.

Tab. 2.3 - Descrição dos pinos do ATmega328.

PINOS DE ALIMENTAÇÃO	
VCC	Tensão de alimentação.
AVCC	Pino para a tensão de alimentação do conversor AD. Deve ser externamente conectado ao VCC, mesmo se o ADC não estiver sendo utilizado.
AREF	Pino para a tensão de referência analógica do conversor AD.
GND	Terra.

PORTB	
PB0	ICP1 – entrada de captura para o Temporizador/Contador 1. CLKO – saída de <i>clock</i> do sistema. PCINT0 – interrupção 0 por mudança no pino.
PB1	OC1A – saída da igualdade de comparação A do Temporizador/Contador 1 (PWM). PCINT1 – interrupção 1 por mudança no pino.
PB2	SS – pino de seleção de escravo da SPI (<i>Serial Peripheral Interface</i>). OC1B – saída da igualdade de comparação B do Temporizador/Contador 1 (PWM). PCINT2 – interrupção 2 por mudança no pino.
PB3	MOSI – pino mestre de saída e escravo de entrada da SPI. OC2A – saída da igualdade de comparação A do Temporizador/Contador 2 (PWM). PCINT3 – interrupção 3 por mudança no pino.
PB4	MISO – pino mestre de entrada e escravo de saída da SPI. PCINT4 – interrupção 4 por mudança no pino.
PB5	SCK – pino de <i>clock</i> da SPI. PCINT5 – interrupção 5 por mudança no pino.
PB6	XTAL1 – entrada 1 do oscilador ou entrada de <i>clock</i> externa. TOSC1 – entrada 1 para o oscilador do temporizador (RTC). PCINT6 – interrupção 6 por mudança no pino.
PB7	XTAL2 – entrada 2 do oscilador. TOSC2 – entrada 2 para o oscilador do temporizador (RTC). PCINT7 – interrupção 7 por mudança no pino.
PORTC	
PC0	ADC0 – canal 0 de entrada do conversor AD. PCINT8 – interrupção 8 por mudança no pino.
PC1	ADC1 – canal 1 de entrada do conversor AD. PCINT9 – interrupção 9 por mudança no pino.
PC2	ADC2 – canal 2 de entrada do conversor AD. PCINT10 – interrupção 10 por mudança no pino.
PC3	ADC3 – canal 3 de entrada do conversor AD. PCINT11 – interrupção 11 por mudança no pino.
PC4	ADC4 – canal 4 de entrada do conversor AD. SDA – entrada e saída de dados da interface a 2 fios (TWI – I2C). PCINT12 – interrupção 12 por mudança no pino.
PC5	ADC5 – canal 5 de entrada do conversor AD. SCL – <i>clock</i> da interface a 2 fios (TWI – I2C). PCINT13 – interrupção 13 por mudança no pino.
PC6	RESET – pino de inicialização. PCINT14 – interrupção 14 por mudança no pino.

PORTD	
PD0	RXD – pino de entrada (leitura) da USART. PCINT16 – interrupção 16 por mudança no pino.
PD1	TXD – pino de saída (escrita) da USART. PCINT17 – interrupção 17 por mudança no pino.
PD2	INT0 – entrada da interrupção externa 0. PCINT18 – interrupção 18 por mudança no pino.
PD3	INT1 – entrada da interrupção externa 1. OC2B – saída da igualdade de comparação B do Temporizador/Contador 2 (PWM) PCINT19 – interrupção 19 por mudança no pino.
PD4	XCK – <i>clock</i> externo de entrada e saída da USART. T0 – entrada de contagem externa para o Temporizador/Contador 0. PCINT 20 – interrupção 20 por mudança no pino.
PD5	T1 – entrada de contagem externa para o Temporizador/Contador 1. OC0B – saída da igualdade de comparação B do Temporizador/Contador 0 (PWM). PCINT 21 – interrupção 21 por mudança no pino.
PD6	AIN0 – entrada positiva do comparador analógico. OC0A – saída da igualdade de comparação A do Temporizador/Contador 0 (PWM). PCINT 22 – interrupção 22 por mudança no pino.
PD7	AIN1 – entrada negativa do comparador analógico. PCINT 23 – interrupção 23 por mudança no pino.

Como pode ser visto na tab. 2.3, os pinos do ATmega são organizados em conjuntos denominados: PORTB, PORTC e PORTD. Cada um deles possui 8 pinos (com exceção do PORTC) organizados pelos nomes: PB0-7, PC0-6 e PD0-7. Cada PORT é um porta bidirecional de I/O de 8 bits com resistores internos de *pull-up* selecionáveis para cada bit. Os registradores de saída possuem características simétricas com capacidade de fornecer e receber corrente, suficiente para o acionamento direto de cargas de até 40 mA. Outra característica importante é que todos os pinos apresentam pelo menos duas funções distintas, até mesmo o pino de *reset* pode ser utilizado como pino de I/O.

2.3 SISTEMA DE CLOCK

Na fig. 2.10, é apresentado o diagrama esquemático do sistema de *clock* do AVR e sua distribuição. Para a redução do consumo de potência, os módulos de *clock* podem ser suspensos usando diferentes modos de programação. O AVR suporta as seguintes opções de *clock*: cristal ou ressonador cerâmico externo, cristal de baixa frequência externo, sinal de *clock* externo e oscilador RC interno, como é exemplificado na fig. 2.11.

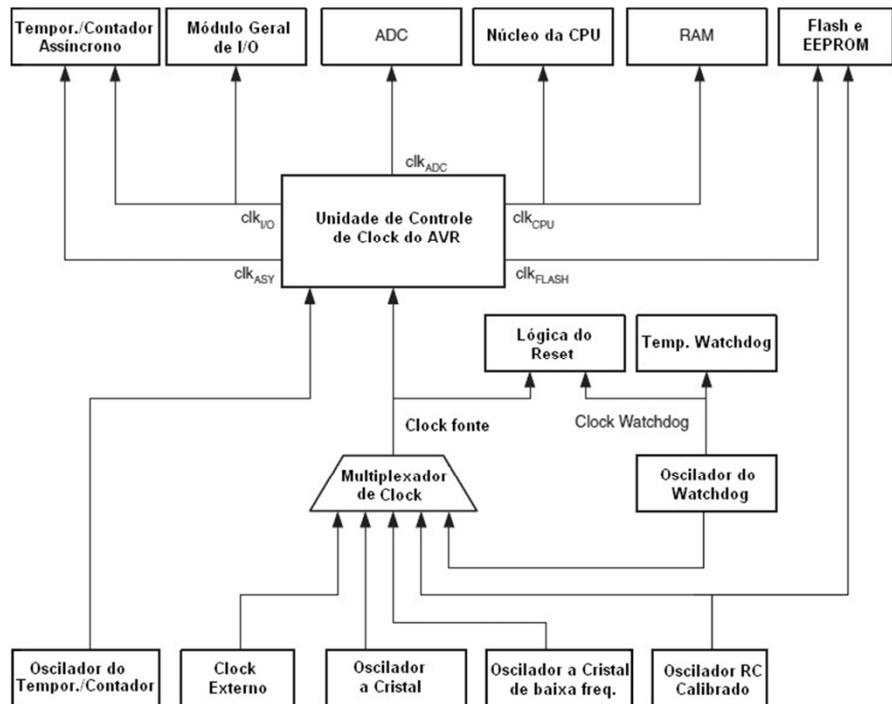


Fig. 2.10- Diagrama esquemático do sistema de *clock* do AVR e sua distribuição.

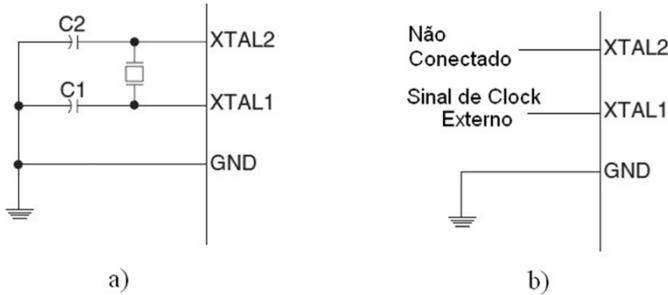


Fig. 2.11- Opções de *clock* externo para o AVR: a) cristal e b) sinal externo.

Quando não é necessário o emprego de um cristal ou ressonador cerâmico externo, é possível utilizar o oscilador interno, permitindo o uso dos pinos XTAL1 e XTAL2 do ATmega328 para outras funções. O oscilador interno pode ser programado para operar com a frequência máxima de 8 MHz (o valor *default* é 1 MHz). Apesar de ter sua frequência dependente da tensão e da temperatura, a frequência do oscilador interno pode ser precisamente calibrada pelo usuário.

2.4 O RESET

A inicialização é fundamental para o trabalho do microcontrolador. Ela é feita durante a energização do circuito ou quando se deseja inicializar os registradores da CPU. Durante o *reset* (inicialização), todos os registradores de entrada e saída são ajustados para os seus valores *default* (padrão) e o programa começa a ser executado a partir do vetor de *reset* (endereço 0 da memória de programa, ou do endereço do *boot loader*). O diagrama do circuito de *reset* é apresentado na fig. 2.12.

As portas de I/O são imediatamente inicializadas quando uma fonte de *reset* é ativa, isso não exige qualquer sinal de *clock*. Após o *reset* ficar inativo, é efetuado um atraso interno automático (configurável) mantendo o *reset* por um pequeno período de tempo. Assim, a tensão de alimentação

pode alcançar um nível estável antes do microcontrolador começar a operar.

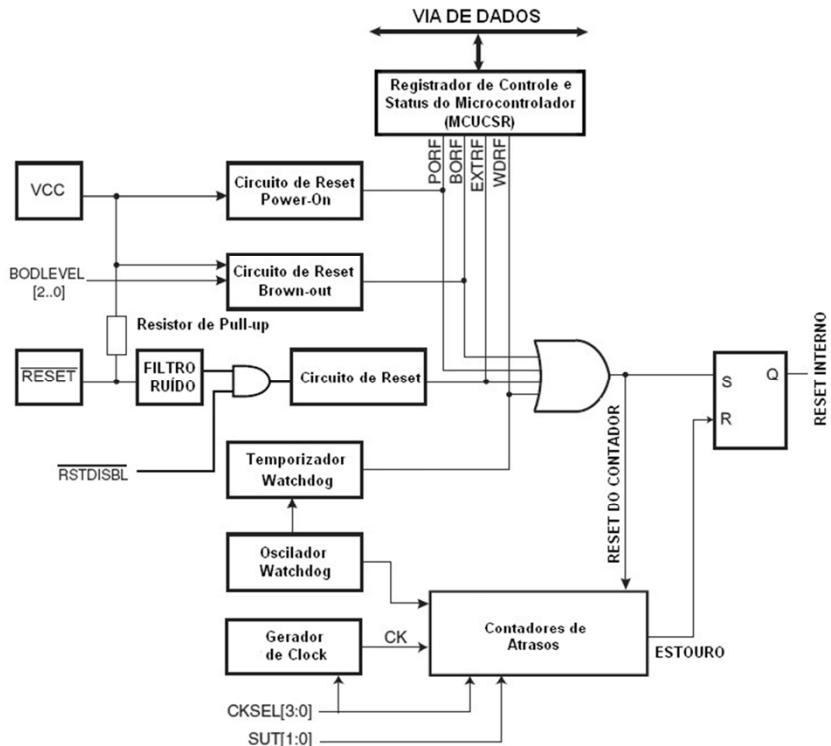


Fig. 2.12- Diagrama do circuito de *reset* do AVR.

O ATmega possui 4 fontes de *reset*:

- *Power-on Reset*: ocorre na energização enquanto a fonte de alimentação estiver abaixo da tensão limiar de *power-on reset* (V_{POT}).
- *Reset externo*: ocorre quando o pino de *reset* é aterrado (0 V) por um determinado período de tempo.
- *Watchdog Reset*: ocorre quando o *watchdog* está habilitado e o seu contador atinge o valor limite.

- Brown-out Reset: ocorre quando a tensão de alimentação cair abaixo do valor definido para o *brown-out reset* (V_{BOT}) e o seu detector estiver habilitado.

Recomenda-se conectar o circuito da fig. 2.13 ao pino de *reset*. Os valores usuais são de $10\text{ k}\Omega$ para o resistor e de 100 nF para o capacitor⁹. Para economia de componentes, o pino de *reset* pode ser ligado diretamente ao VCC. Todavia, o *reset* físico é indicado quando se deseja inicializar manualmente o microcontrolador. Assim, no circuito da fig. 2.13 pode-se empregar um botão entre o pino de *reset* e o terra.

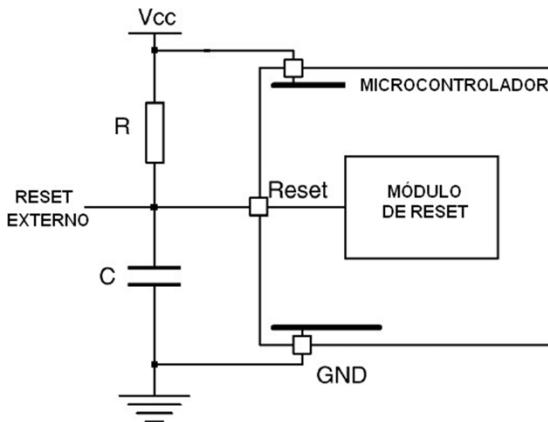


Fig. 2.13- Circuito para ligação ao pino de *reset*.

⁹ Consultar o *application note* do fabricante, AVR042: AVR *Hardware Design Considerations*.

2.5 GERENCIAMENTO DE ENERGIA E O MODO SLEEP

Para a economia de energia, necessária com o uso de baterias, o modo *sleep* (soneca) permite o desligamento de partes internas do chip não utilizadas do microcontrolador. Esse modo é tão importante que existe um código de instrução para ele: o SLEEP. O AVR possui 6 modos possíveis de *sleep*:

- *Idle*: a atividade da CPU é suspensa, mas a SPI, a USART, o comparador analógico, o ADC, a interface serial à 2 fios, os temporizadores/contadores, o *watchdog timer* e o sistema de interrupção continuam operando. Basicamente, o *clock* da CPU e da memória *flash* são suspensos.
- *Redução de Ruído para o ADC*: a CPU é parada, mas continuam operando o ADC, as interrupções externas, igualdade de endereço da interface serial à 2 fios, o temporizador/contador 2 e o *watchdog timer* (se habilitado). Esse modo é empregado para reduzir o ruído para o ADC e garantir sua resolução.
- *Power-down*: o funcionamento do oscilador externo é suspenso, mas continuam operando a interface serial à 2 fios, as interrupções externas e o *watchdog timer* (se habilitado). Basicamente, interrompe todos os *clocks* gerados.
- *Power-save*: igual ao modo *power-down*, com exceção que o contador/temporizador 2 continua trabalhando assincronamente.
- *Standby*: é idêntico ao modo *power-down*, com exceção que o oscilador é mantido funcionando (válido para oscilador externo a cristal ou ressonador cerâmico). O microcontrolador ‘desperta’ do *sleep* em 6 ciclos de *clock*.
- *Extended Standby*: idêntico ao modo *power-save*, com exceção que o oscilador é mantido funcionando. Leva 6 ciclos de *clock* para o microcontrolador ‘despertar’.

De acordo com os requisitos do projeto, um ou outro modo de economia de energia deverá ser escolhido; quanto maior o número de fontes de *clock* desligadas, menor o consumo de corrente. É importante consultar o manual do fabricante para compreender como se obter o máximo desempenho. Na tab. 2.4, são apresentadas quais fontes são desligadas e quais sinais ‘despertam’ o microcontrolador.

Tab. 2.4 – Fontes desligadas e sinais de ‘despertar’ para os diferentes modos *sleep*.

Modos Sleep	Sinais de <i>clock</i> ativos					Osciladores		Fontes de <i>wake-up</i> (para ‘despertar’)						BOD ⁴ desabilitado por software	
	Clock CPU	Clock FLASH	Clock IO	Clock ADC	Clock ASY	Fonte principal de <i>clock</i> habilitada	Oscilador do Temporizador habilitado	INT1, INT0 e mudança nos pinos	Casamento de endereço da interface serial à 2 fios	Temporizador 2	SRAM/EEROM prontos	ADC	Watchdog Timer	Outras I/O	
<i>Idle</i>		X	X	X	X	X ²	X ³	X	X	X	X	X	X	X	
Redução de Ruído para o ADC			X	X	X	X ²	X ³	X	X ²	X	X	X	X		
<i>Power-Down</i>							X ³	X					X		X
<i>Power-Save</i>					X		X ²	X ³	X	X			X		X
<i>Standby</i> ¹					X		X ³	X					X		X
<i>Extended Standby</i>				X ²	X	X ²	X ³	X	X				X		X

Notas:

1 – Recomendado somente quando se emprega cristal externo ou ressonador.

2 – Se o Temporizador/Contador 2 estiver no modo assíncrono.

3 – Somente para as interrupções INT0 e INT1 por nível.

4 – BOD somente em dispositivos com *picoPower* (ATmega48PA/88PA/168PA/328P).

3. SOFTWARE E HARDWARE

O uso de softwares é imprescindível quando se trabalha com microcontroladores. O código fonte (programa ou firmware) para o microcontrolador necessita ser escrito, compilado, depurado e gravado. Todas essas tarefas são realizadas com o suporte de softwares adequados. Assim, neste capítulo é feita uma introdução aos programas necessários para o trabalho com o AVR e de um programa de gravação para uso com a plataforma Arduino. Também é sugerido o uso de um software de simulação, o Proteus® (ISIS).

Este capítulo é introdutório e deve ser consultado quando for conveniente. Seu objetivo é uma introdução rápida ao AVR Studio®, ao Arduino e ao Proteus (ISIS), suficiente para o início dos trabalhos com os microcontroladores AVR. O aprendizado das demais características dos softwares será decorrente do seu uso.

Para se escrever o código fonte e gravar os microcontroladores AVR é comum o emprego do programa AVR Studio, obtido gratuitamente no sítio da Atmel (www.atmel.com). Para a programação em C, nas versões anteriores ao AVR Studio 5, é necessário instalar explicitamente o seu *plug-in*, o programa WinAVR, o qual possui um módulo compilador (AVR-GCC) e depurador (*debugger*); além de um módulo independente para a gravação, o AVRdude. O WinAVR também é gratuito e pode ser obtido no sítio <http://winavr.sourceforge.net/>. Existem outras opções de compiladores C, que por serem pagos, não serão abordados.

Para a gravação dos microcontroladores deve-se adquirir um hardware específico. A Atmel, por exemplo, produz o AVR *Dragon*, com capacidade de emulação e *debug in system* (depuração no sistema físico) possuindo interface JTAG. Outras opções estão disponíveis comercialmente e existem esquemas para *download* na internet. Se for utilizada a plataforma Arduino, essa possui o seu próprio hardware de gravação.

Uma ferramenta de análise disponível no AVR Studio é o depurador, que é importante para achar erros e compreender o funcionamento do código. Entretanto, avaliar o comportamento real do microcontrolador exige a montagem em hardware. Outra saída, menos onerosa e que permite um rápido desenvolvimento, é o emprego de um software de simulação. O Proteus (ISIS schematic capture) produzido pela *Labcenter Electronics* (www.labcenter.co.uk) simula inúmeros microcontroladores, incluindo vários AVRs. É possível adquirir diferentes licenças para diferentes microcontroladores. O Proteus também possui o programa ARES PCB Layout, permitindo a confecção da placa de circuito impresso diretamente do diagrama esquemático do circuito simulado. No sítio da Labcenter é possível baixar uma versão de demonstração do Proteus, que permite a escrita de programas para rodar nos seus exemplos disponíveis e dispõe de todas as ferramentas do programa profissional. Todavia, não é possível salvar, imprimir ou projetar novos circuitos.

Para o trabalho exclusivamente com a plataforma Arduino pode ser empregado o seu próprio software gratuito (www.arduino.cc). Ele não permite o *debug* do programa e não é adequado ao desenvolvimento profissional de projetos. Entretanto, o seu ambiente integrado de desenvolvimento – IDE tem grande aceitação para desenvolvimentos rápidos e que não demandem muito conhecimento técnico e desempenho do microcontrolador. Todavia, dado o caráter técnico deste livro, o AVR Studio é priorizado e a IDE do Arduino é apresentada de forma sucinta no capítulo 22.

3.1 A PLATAFORMA ARDUINO

Para a avaliação real dos programas apresentados foi utilizada a plataforma Arduino. Essa plataforma permite a gravação direta do microcontrolador quando ligada a uma porta USB de um computador. Suas vantagens são: não exigir o uso de gravador dedicado, apresentar tamanho compacto e um grande conjunto de placas auxiliares no formato

de módulos, chamadas de *shields*, que facilitam em muito o desenvolvimento de projetos.

Uma característica interessante do Arduino é o seu conjunto de barras de pinos soquete que permitem conectar facilmente os *shields*, tornando fácil montar diversos sistemas.

O Arduino é uma plataforma de prototipação eletrônica de código aberto (*open-source*), qualquer pessoa pode produzi-la, é fácil de programar e usar, além de possuir preço acessível e ser facilmente encontrada.

O modelo de Arduino utilizado neste trabalho é o Arduino UNO, que é o modelo mais atual e utilizado, uma melhoria da versão Duemilenove¹⁰ (ver a fig. 3.1). Ambos são muito similares e podem ser utilizados indistintamente com os exemplos deste livro. O Arduino UNO emprega o microcontrolador ATmega328P, com a disponibilização de 20 pinos de I/O, trabalha com um cristal de 16 MHz, possui conexão USB, conector de potência - *jack*, botão de *reset* e disponibiliza pinos para a gravação *In-System*. Pode ser alimentado pela USB, por fonte de tensão ou mesmo bateria. Na fig. 3.2, é apresentado o seu circuito completo.

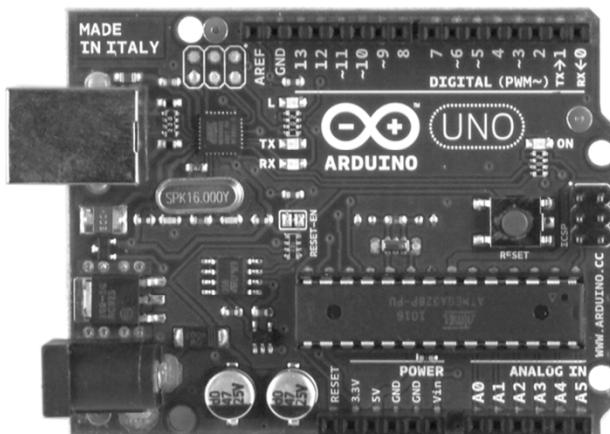


Fig. 3.1- Arduino UNO (fonte: www.arduino.cc).

¹⁰ Existem vários modelos de Arduinos, para maiores referências consultar: www.arduino.cc.

Arduino™ UNO Reference Design

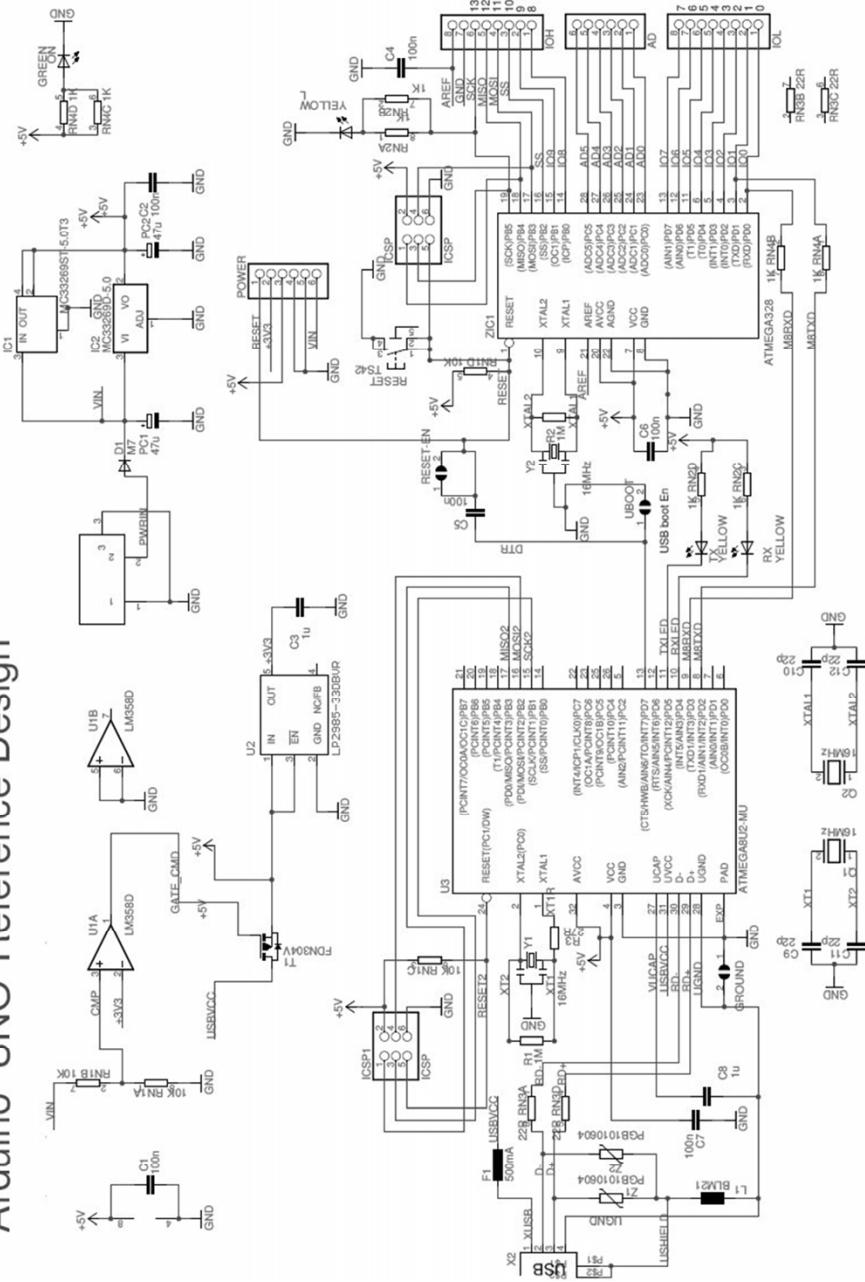


Fig. 3.2 – Circuito do Arduino UNO (fonte: www.arduino.cc).

Na tab. 3.1, é apresentada a correlação entre os nomes dos pinos do Arduino e os nomes dos pinos do ATmega328. Essa informação é fundamental na hora da programação, pois na placa de circuito impresso do Arduino os pinos do microcontrolador não possuem seus nomes originais.

Tab. 3.1 – Correlação entre os pinos do Arduino e do ATmega328.

Arduino	ATmega328	Arduino	ATmega328	Arduino	ATmega328
<i>Analog In</i>	PORTC		PORTD		PORTB
A0	PC0	0	PD0	8	PB0
A1	PC1	1	PD1	9	PB1
A2	PC2	2	PD2	10	PB2
A3	PC3	3	PD3	11	PB3
A4	PC4	4	PD4	12	PB4
A5	PC5	5	PD5	13	PB5
		6	PD6		
		7	PD7		

3.2 CRIANDO UM PROJETO NO AVR STUDIO

Após instalar o *AVR Studio*¹¹, é hora de criar o primeiro projeto. Esse conterá o arquivo do código fonte e todos os demais necessários para o trabalho com o microcontrolador.

Para iniciar um novo projeto pode ser utilizado o menu <File> <New> e <Project> ou pode-se clicar diretamente em <New Project> da janela <Start Page>, a qual aparecerá automaticamente como opção *default* da inicialização do *AVR Studio* (fig. 3.3). Será aberta uma janela, na qual se deve clicar em <AVR Assembly>, para a programação em *Assembly*, ou em <AVR GCC> <C> <C Executable Project>, para programação na linguagem C (fig. 3.4). Caso algum projeto já tenha sido aberto anteriormente, ele aparecerá nessa janela. Também deve ser preenchido o nome do programa e indicado o local para o salvamento dos arquivos.

¹¹ Neste livro é utilizado o *AVR Studio* 5.1, que já instala todas as ferramentas necessárias a programação. Caso se utilize o *AVR Studio* 4 é necessário primeiro instalar o *WinAVR*.

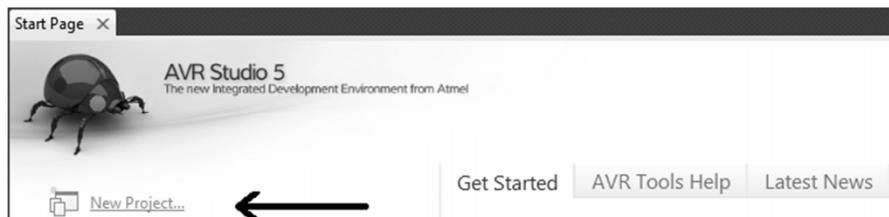


Fig. 3.3 – Criando um novo projeto na janela <Start Page>.

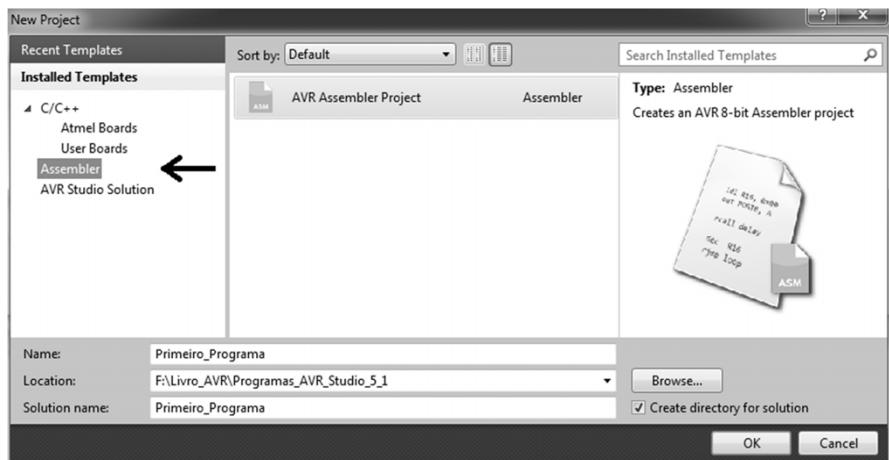


Fig. 3.4 – Janela <New Project>.

Após a execução dos passos supracitados, deve-se clicar em <OK>; irá aparecer a janela da figura 3.5, onde se escolhe qual microcontrolador se programará, no caso do Arduino Uno, o ATmega328P. Após essa escolha, clica-se novamente em <OK>, se aparecer uma janela para a seleção do depurador, deve-se selecionar <AVR Simulator>, ou se utilizar a combinação de teclas <Alt + F7>, fig. 3.6, opção <Debugging> <Selected Debugger> <AVR Simulator>. Essa janela possui várias configurações, incluindo a escolha do microcontrolador e será utilizada com frequência.

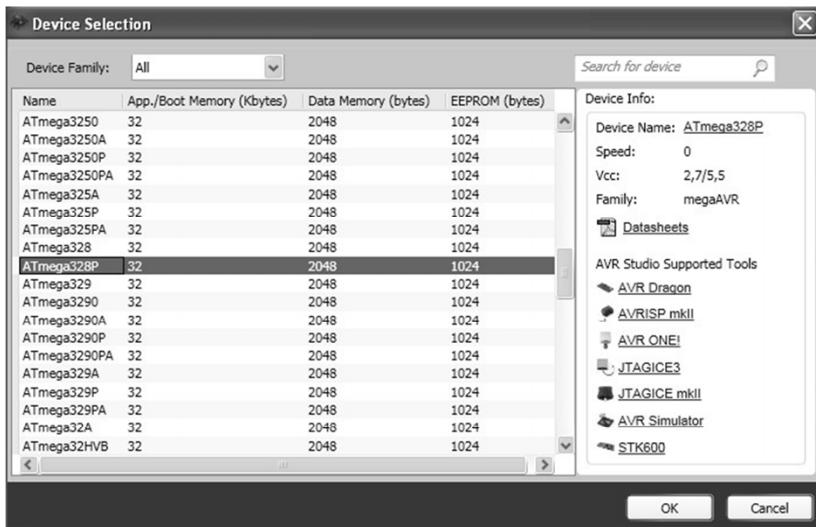


Fig. 3.5 – Definindo o tipo de programação e o nome do projeto.

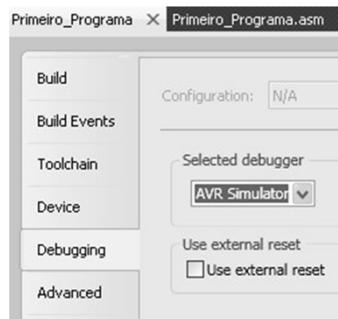


Fig. 3.6 – Escolhendo a ferramenta de depuração.

Após as configurações iniciais, o programa já pode ser escrito na janela em branco, com o nome aparecendo na aba superior, neste caso, Primeiro_Programa.asm. Na direita da tela de programa irá aparecer os arquivos relativos ao projeto e suas dependências (fig. 3.7).

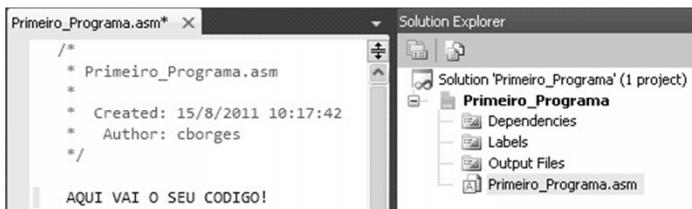


Fig. 3.7 – Janela para a escrita do programa.

Com o programa escrito, já é possível montá-lo¹². Isso é feito clicando-se no ícone <Build nome> ou <Build Solution> (F7) (ver a fig. 3.8). Na parte inferior da janela do AVR Studio aparecerá as mensagens do compilador e o número de bytes utilizados pelo programa (fig. 3.9).

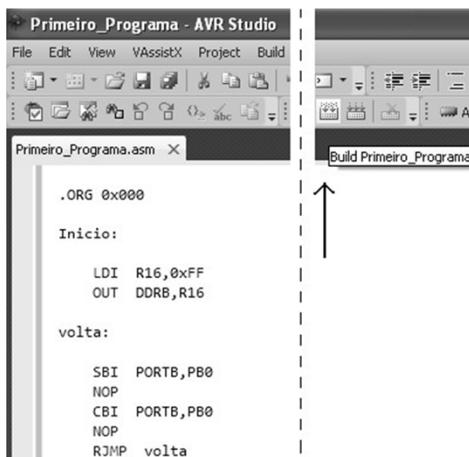


Fig. 3.8 – Compilando o programa.

```
ATmega328P memory use summary [bytes]:
Segment Begin End Code Data Used Size Use%
-----
[.cseg] 0x000000 0x0000e 14 0 14 32768 0.0%
[.dseg] 0x000100 0x000100 0 0 0 2048 0.0%
[.eseg] 0x000000 0x000000 0 0 0 1024 0.0%
Assembly complete, 0 errors. 0 warnings
```

Fig. 3.9 – Resultado da compilação.

¹² O ato de compilar gera um arquivo em *assembly*. A geração do arquivo *.hex ocorre no processo de montagem. Entretanto, neste livro, dependendo do contexto, a palavra *compilar* pode ter sido empregada como sinônimo para o processo completo de montagem.

Uma vez que o programa esteja pronto e compilado, é possível fazer sua depuração. Os ícones relativos ao trabalho com a depuração são apresentados na fig. 3.10.

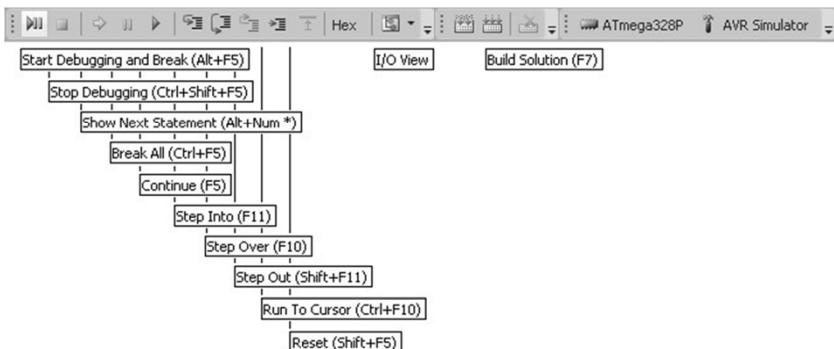


Fig. 3.10 – Ícones para o trabalho com a depuração do programa.

Na depuração, os registradores do microcontrolador podem ser vistos, incluindo variáveis e outros detalhes referentes ao funcionamento do microcontrolador; para a devida seleção é utilizado o ícone <I/O View>, figs. 3.10-11. Uma possível seleção dos registradores de entrada e saída do microcontrolador é apresentada na fig. 3.11. A execução do programa começa com o <Start Debugging and Break> (Alt + F5) e cada linha de código pode ser executada com o <Step Into> (F11). O AVR Studio permite, ainda, entrar e sair de sub-rotinas, executar o programa até um ponto de parada (*breakpoint*) e outras características usuais empregadas na depuração. Os valores dos bits de um registrador são apresentados em pequenos quadrados (fig. 3.11). A escrita neles é feita com um clique do mouse. Seu uso permite a alteração dos registradores de entrada e saída do microcontrolador e a simulação de estímulos externos.

Quando se programa com a linguagem C, é importante configurar a otimização utilizada para o compilador. Para isso, pode ser utilizada a combinação (Alt + F7), ou clicar-se na opção <nome_programa Properties ... Alt + F7> no menu <Project>. Essa opção só estará disponível quando se

cria um novo projeto em C (fig. 3.4). Na janela aberta (fig. 3.12) deve-se selecionar <Toolchain> na lateral esquerda e depois a opção <AVR/GNU C compiler - Optimization>, onde se escolhe a opção <Optimize for Size(-Os)>.

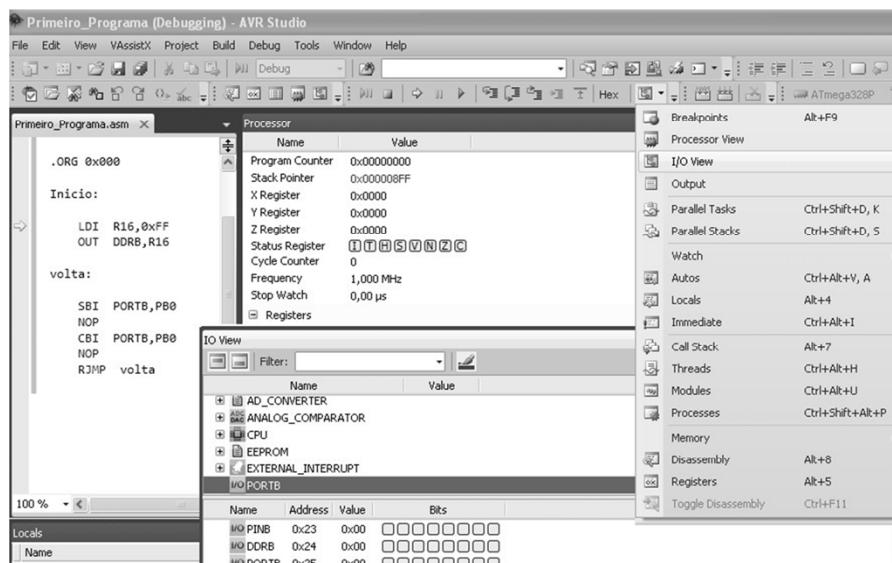


Fig. 3.11 – Depurando o programa.

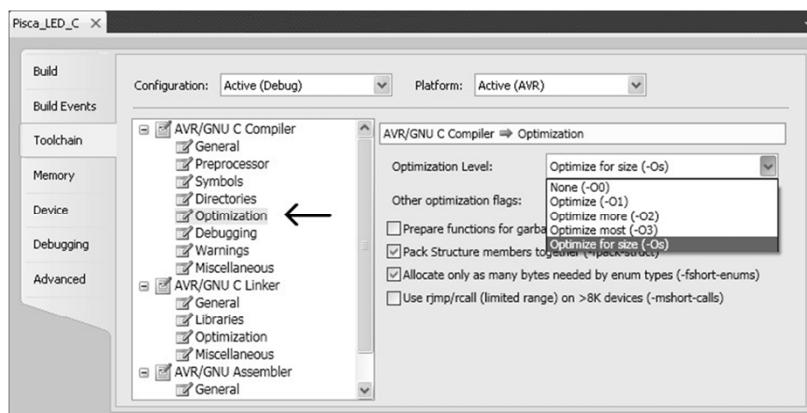


Fig. 3.12 – Configurando a otimização para o um programa em C.

3.3 SIMULANDO NO PROTEUS (ISIS)

Uma vez que o programa para o microcontrolador esteja pronto e montado, ele pode ser simulado no Proteus. Nesse caso, basta desenhar o circuito e carregar o arquivo *.hex para o microcontrolador, como apresentado a seguir.

O circuito a ser simulado deve ser simplificado sempre que possível, eliminando componentes externos dispensáveis para o teste do programa. Isso facilita a simulação e diminui erros. Para criar um circuito microcontrolado no ISIS, clique em <New Design> no menu <File> (fig. 3.13).



Fig. 3.13 – Criando um circuito microcontrolado no Proteus.

Os componentes eletrônicos¹³ disponíveis podem ser encontrados no ícone ou no (fig. 3.14), que abre uma janela com todas as bibliotecas de componentes disponíveis (fig. 3.15). Nessa janela, é apresentado o componente e seu encapsulamento, se disponível. Também existe um campo para a procura de componentes por palavras chave (*keywords* - fig. 3.15).

¹³ O ISIS não apresenta os pinos de alimentação dos circuitos integrados.

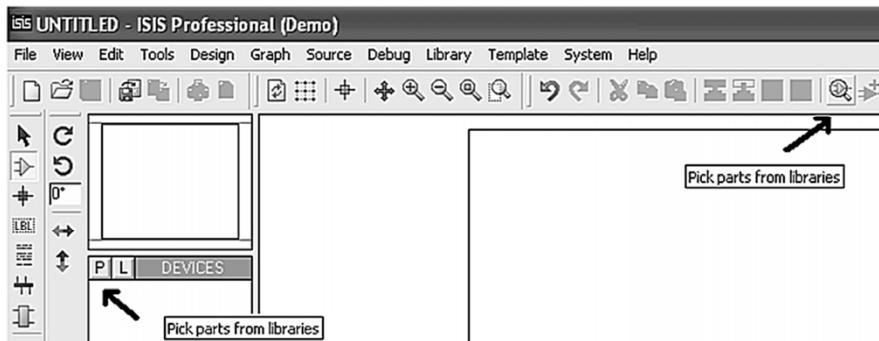


Fig. 3.14 – Encontrando os componentes eletrônicos no Proteus.

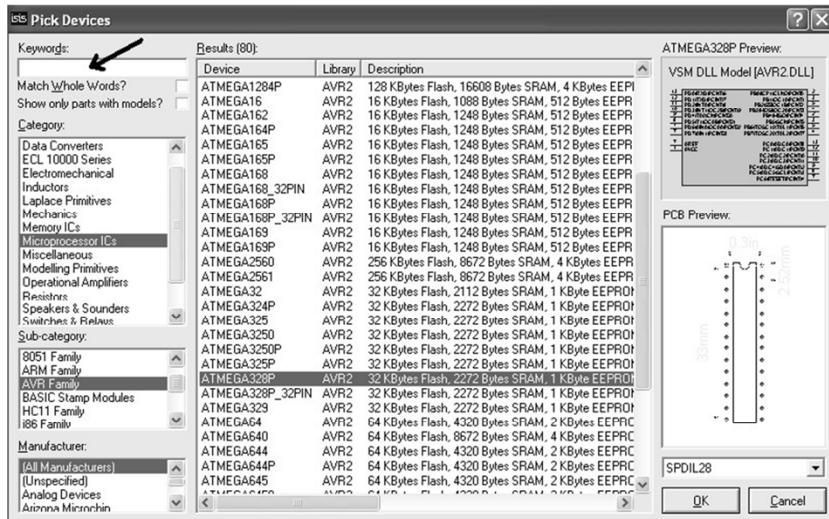


Fig. 3.15 – Janela de procura de componentes do Proteus.

No ícone da barra lateral esquerda se encontram os símbolos de terra e de alimentação. A conexão entre os componentes do circuito é feita com o emprego do ícone , conforme fig. 3.16. Após a conclusão do diagrama esquemático, basta um duplo clique sobre o microcontrolador para que se abra uma janela (fig. 3.17). Nela se deve indicar a localização

do arquivo ***.hex**, o qual possui o código a ser executado (criado na montagem do programa pelo *AVR Studio*) e se definem as demais configurações de trabalho do microcontrolador (ver o capítulo 23). No exemplo da fig. 3.17, está definido o emprego de um oscilador externo de 16 MHz. Os valores *default* não necessitam ser alterados, e inicialmente, não existe a necessidade de preocupação com eles. A única a alteração é na forma e na frequência de trabalho do microcontrolador, quando houver necessidade.

Após a configuração do microcontrolador, basta clicar no ícone (*play*) da barra horizontal inferior do canto esquerdo (fig. 3.18), então, o programa contido no arquivo ***.hex** será executado. Se tudo estiver correto, nenhuma mensagem de erro será gerada e a simulação ocorrerá. Os ícones à direita do *play* permitem reiniciar, pausar e parar a simulação, respectivamente.

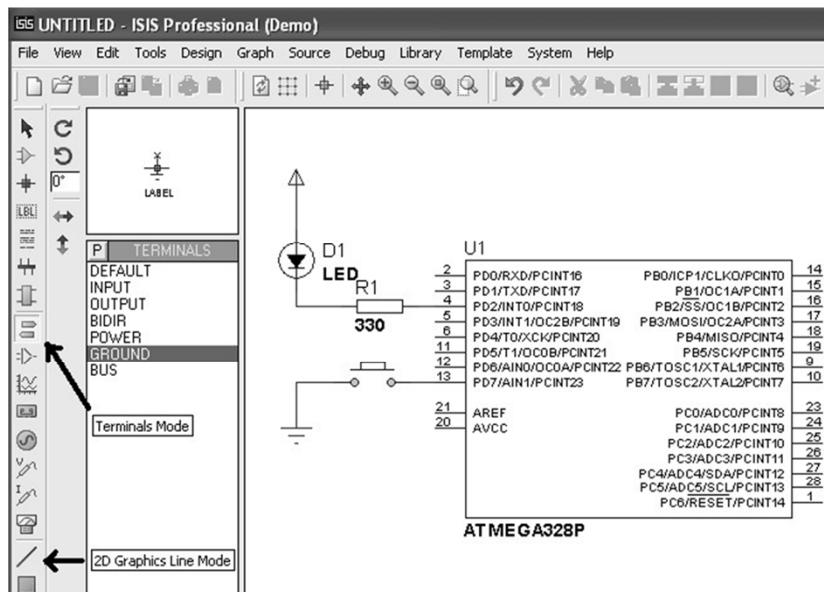


Fig. 3.16 – Ícones para o terra, a alimentação e fio de conexão.

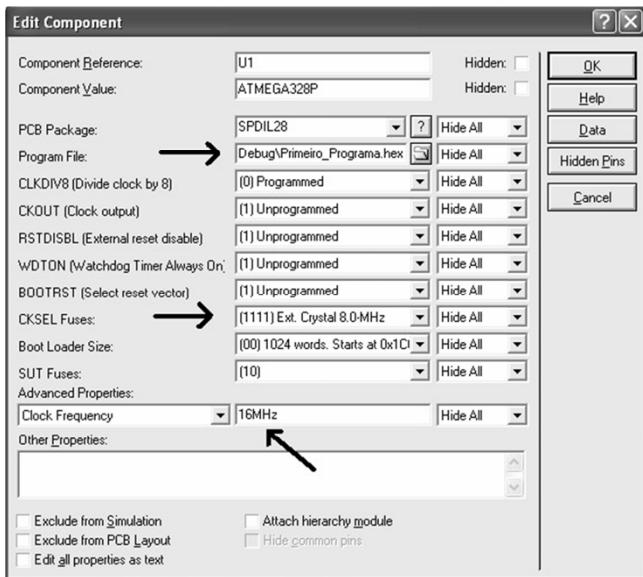


Fig. 3.17 – Janela para definir o arquivo de programa e outros detalhes referentes ao microcontrolador.

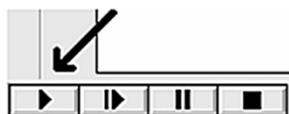


Fig. 3.18– Executando a simulação do circuito.

3.4 GRAVANDO O ARDUINO

Uma vez desenvolvido e montado, o programa precisa ser gravado no microcontrolador. No Arduino, isso pode ser feito diretamente no seu ambiente integrado de desenvolvimento (IDE, capítulo 22). Entretanto, neste livro o *AVR Studio* é utilizado, priorizando o aspecto técnico e de depuração para a programação.

Como o *AVR Studio* não permite a gravação direta do Arduino, é necessário empregar um programa específico com essa finalidade. A solução comumente empregada é o *AVRdude*, utilizada inclusive pela IDE

do Arduino. Para facilitar a gravação emprega-se uma interface gráfica em conjunto com o AVRdude. Uma boa escolha é o AVR8 Burn-O-Mat, gratuito, fácil de instalar, configurar e usar, que pode ser baixado de <http://avr8-burn-o-mat.aaabbb.de/>. Para instalá-lo, é necessário o Java SE Runtime Environment (JRE), que pode ser baixado gratuitamente de <http://java.sun.com/javase/downloads/index.jsp>. O AVRdude encontra-se nos arquivos de programa do Arduino, baixados de <http://arduino.cc/en/Main/Software>. Na fig. 3.19, é apresentada a janela principal do AVR8 Burn-O-Mat; nela se indica qual o microcontrolador que será gravado, o arquivo de programa (*.hex) e, se desejado, o arquivo para gravação da memória EEPROM (*.eep, ver o capítulo 23). Existe um espaço em branco destinado as mensagens do AVRdude na gravação.

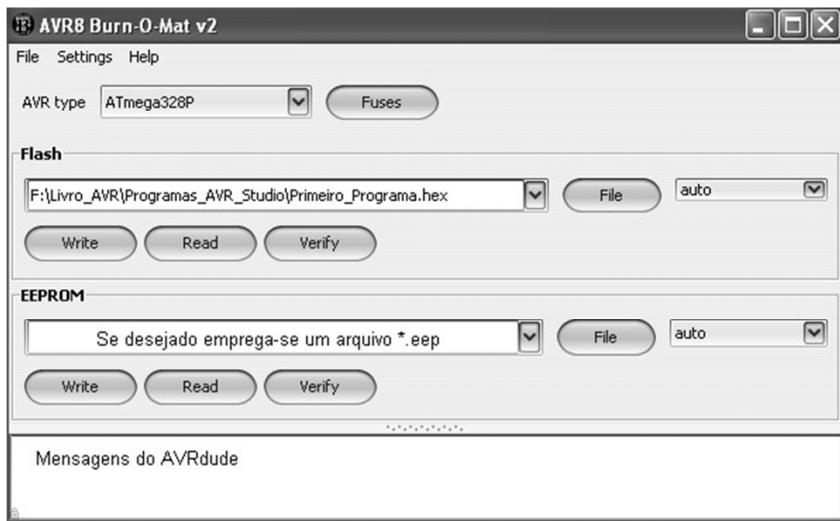


Fig. 3.19– Programa para gravar o Arduino.

O primeiro passo a ser realizado é a configuração do AVRdude no menu <Settings> <AVRDUEDE>, ação que leva a abertura da janela da fig. 3.20.

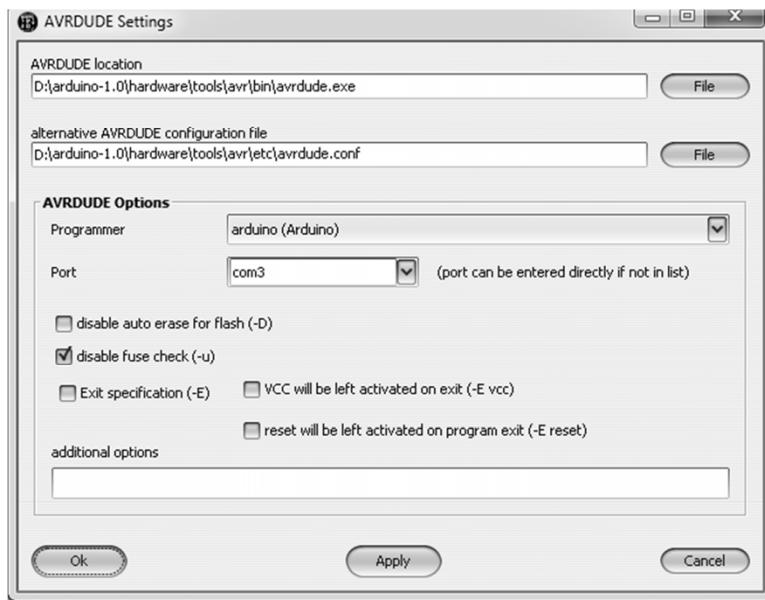


Fig. 3.20– Configurando o AVRdude no AVR8 Burn-O-Mat.

Na janela (fig. 3.20), é informado em <AVRDUDE Location> onde está o programa avrdude.exe que será utilizado. Ele encontra-se na pasta do Arduino, que deve ter sido baixada do sítio citado anteriormente. Deve ser indicado o seu caminho, neste caso:

D:\arduino-1.0\hardware\tools\avr\bin\avrdude.exe

Depois, indica-se onde se encontra o arquivo de configuração em <alternative AVRDUDE configuration file>:

D:\arduino-1.0\hardware\tools\avr\etc\avrdude.conf

Após, deve ser informado em <Programmer> qual o tipo de programador (hardware) será empregado. Para o Arduino, a escolha é:

arduino (Arduino)

Obs.: se for utilizada uma versão da IDE do Arduino inferior a 1.0, o programador será:

stk500v1(Atmel STK500 Version 1.x firmware)

Neste ponto, espera-se que o Arduino já tenha sido instalado no seu computador. Quando isso ocorre, o sistema operacional lhe atribui uma porta COM virtual (ver a seção 15.3.); recomenda-se que ela esteja entre as COM1-COM8. É necessário especificar qual é a porta COM do Arduino em <Port>. Caso necessário essa informação pode ser encontrada no painel de controle do Windows, onde é possível também alterar sua numeração. Após esses passos clica-se em <Apply> e depois em <OK>. O programa volta a tela principal e a gravação pode ser efetuada.

Outro detalhe importante é o botão <Fuses> da fig. 3.19, que permite alterar algumas configurações para o trabalho do microcontrolador, como por exemplo, qual o oscilador será empregado. O detalhamento deles para ATmega328 será feito no capítulo 23. Eles não devem ser alterados para o Arduino, pois já vêm configurados de fábrica e corre-se o risco de se impedir a gravação do microcontrolador. Os fusíveis só devem ser alterados quando se souber exatamente o que se está fazendo.

4. PROGRAMAÇÃO¹⁴

Neste capítulo, é feita uma breve revisão sobre os conceitos fundamentais para a programação de microcontroladores. São apresentadas algumas técnicas prévias à escrita do programa, tais como: algoritmos, fluxogramas e máquina de estados. Na sequência, são feitas algumas considerações sobre a programação com a linguagem *assembly*, um detalhamento da linguagem de programação C e o seu emprego no ATmega. Este capítulo pode ser consultado quando conveniente e seu objetivo é a supressão de dúvidas, sendo insuficiente para o ensino de programação.

Quando se projeta sistemas microcontrolados, a programação é tão importante quanto o hardware desenvolvido. Um bom programa pode acrescentar funcionalidades extras ao hardware e aumentar consideravelmente o desempenho e confiabilidade do mesmo. O casamento entre hardware e firmware (programa que roda no microcontrolador) deve ser preciso. Em projetos complexos, a equipe se dividirá entre programadores e projetistas de hardware. Logo, para o programador, entender como o hardware funciona é o primeiro passo para uma programação eficiente.

Além de compreender o que deve ser executado pelo firmware, o programador necessita estruturar bem o programa, baseando-se em algoritmos eficientes. Programadores inexperientes, e mesmo alguns que se acham experientes, acreditam que o tempo gasto no desenvolvimento de algoritmos e na forma como o programa deve funcionar (estruturação) é um desperdício, passando diretamente à programação, criando estruturas e resolvendo problemas digitando diretamente linhas de código, sem um raciocínio prévio. O tempo usado no planejamento é um tempo ganho no

¹⁴ Este capítulo contou com a colaboração do prof. Leandro Schwarz.

futuro. Assim, como a construção de um edifício que não começa sem um projeto, o desenvolvimento de um software/firmware deve ser bem planejado. A menos que o programador tenha muita experiência em programação, não se recomenda a programação direta sem pelo menos um rascunho prévio do programa ou das partes fundamentais do mesmo.

Basicamente, o processo de desenvolvimento de um programa deve seguir os seguintes passos:

- Análise do problema a ser solucionado. Quais as exigências que devem ser cumpridas?
- Elaboração dos algoritmos exigidos e estruturação do programa com o uso de fluxogramas ou outras ferramentas, se necessário.
- Escrita do programa na linguagem de programação escolhida.
- Compilação do programa para a linguagem de máquina (entendida pelo microcontrolador).
- Testes (depuração) e correções necessárias.

Cada programador encontrará a técnica de programação que mais lhe agrada e, com a prática, a programação se tornará cada vez mais rápida e eficiente.

4.1 ALGORITMOS

Um algoritmo nada mais é do que a descrição dos passos que devem ser executados por um determinado programa, ou seja, é um conjunto de instruções a serem executadas por um processador, com uma sequência de comandos para resolver um problema específico.

Um algoritmo recebe um ou mais valores de entrada e produz um ou mais valores de saída, indicando os passos a serem seguidos pelo processador na execução de uma determinada função. Quando se cria um algoritmo, deve-se ter noção de como o microcontrolador pode processar as

informações e, assim, pode-se utilizar uma linguagem natural para descrever o que se deseja que ele faça.

O algoritmo a seguir apresenta uma lógica para separar um número decimal em seus dígitos individuais (máximo de 3 dígitos) e imprimir o resultado. Por exemplo, o número 147 seria decomposto nos dígitos individuais 1, 4 e 7.

1. **Zera o vetor DIGITOS;** //DIGITOS[0]=0; DIGITOS[1]=0; DIGITOS[2]=0;
2. **Lê o valor a ser convertido e salva em X;**
3. **Se X for maior que 999 imprime erro e finaliza (passo 8), senão:**
4. **i = 0;** //zera o contador auxiliar
5. **Divide X por 10, guarda o resto em DIGITOS[i] e o inteiro em X;**
6. **i = i + 1;** //incrementa o contador auxiliar
7. **Se X for igual a zero, imprime DIGITOS e finaliza (passo 8), senão volta ao passo 5.**
8. **Fim**

A principal parte do algoritmo acima está na divisão do número a ser convertido por 10, onde o resto é um dos dígitos individuais constituintes do número e o inteiro da divisão é novamente dividido. O processo se repete até restar zero para a divisão.

Todo algoritmo tem um mecanismo para a resolução de um determinado problema, quanto mais inteligente esse mecanismo, menor será o número de comandos e mais rápido será a resolução do problema.

A vantagem de se escrever algoritmos é que eles formam uma linguagem universal dentro das linguagens de programação, permitindo que qualquer código-fonte possa ser facilmente portável para qualquer outra linguagem, ou implementável por qualquer outra pessoa, de forma rápida e segura.

4.1.1 ESTRUTURAS DOS ALGORITMOS

Quando se escreve um algoritmo, pode-se empregar estruturas funcionais que se aproximam das empregadas pelas linguagens de programação normais. Estas estruturas são:

FAÇA ... ENQUANTO

Estrutura de repetição, que realiza determinadas ações até que certa condição não seja mais válida. A ação é realizada uma vez e depois a condição é verificada.

ENQUANTO ... FAÇA

Estrutura de repetição, que realiza determinadas ações até que certa condição não seja mais válida, no entanto, agora a condição é verificada primeiro e a ação depois.

PARA ... DE ... ATÉ ... FAÇA

Mais uma estrutura de repetição, que realiza determinadas ações. Entretanto, nesse caso, a estrutura está voltada para que a ação seja repetida um determinado número de vezes. A condição novamente é primeiro verificada e somente depois a ação é realizada.

SE ... ENTÃO (SENÃO ...) ... FIM SE

Estrutura condicional que realiza determinadas ações se a condição for verdadeira e outras ações caso a condição seja falsa.

FAÇA CASO (CASO ...) SENÃO ... FIM CASO

Estrutura também condicional, que realiza ações para cada situação.

Os operadores utilizados são os seguintes:

ATRIBUIÇÃO (=), atribui um valor a uma variável.

ARITMÉTICOS (+, -, *, /), realizam as quatro operações básicas.

RELACIONAIS (>, <, =, !=), utilizados para descrição de condições.

LÓGICOS (E, OU, NÃO), utilizados para estabelecer uma relação entre condições.

Por exemplo, no algoritmo apresentado anteriormente, para o cômputo dos dígitos individuais de um número, o uso de estruturas resultaria em:

```

PARA n DE 0 ATÉ 2 FAÇA //zera DIGITOS
    DIGITOS[n]=0;
FIM_PARA

X = valor_conversao;      //carrega o valor para conversão em X

SE X>999                  //se o valor for maior que 999 imprime erro e finaliza
    Imprime Erro;

SENAO
    i=0;                      //zera contador auxiliar

FAÇA
    DIGITOS[i]=X%10;          //divide X por 10 e guarda o resto em DIGITOS[i]
    X=X/10;                   //salva em X o inteiro de X dividido por 10
    i=i+1;                    //incrementa o contador auxiliar
ENQUANTO X!=0               //enquanto X é diferente de zero repete o FAÇA

    Imprime DIGITOS;         //imprime o resultado

FIM_SE

FIM

```

4.2 FLUXOGRAMAS

Fluxograma é uma representação gráfica das operações e sequência de encadeamento de um processo. Em programação é empregado para facilitar a criação e o entendimento de um programa.

Existem vários símbolos empregados na construção de fluxogramas; os principais são apresentados na fig. 4.1. Um fluxograma bem estruturado permite a fácil correção de eventuais erros no programa e uma rápida tradução para a linguagem escolhida para a programação.

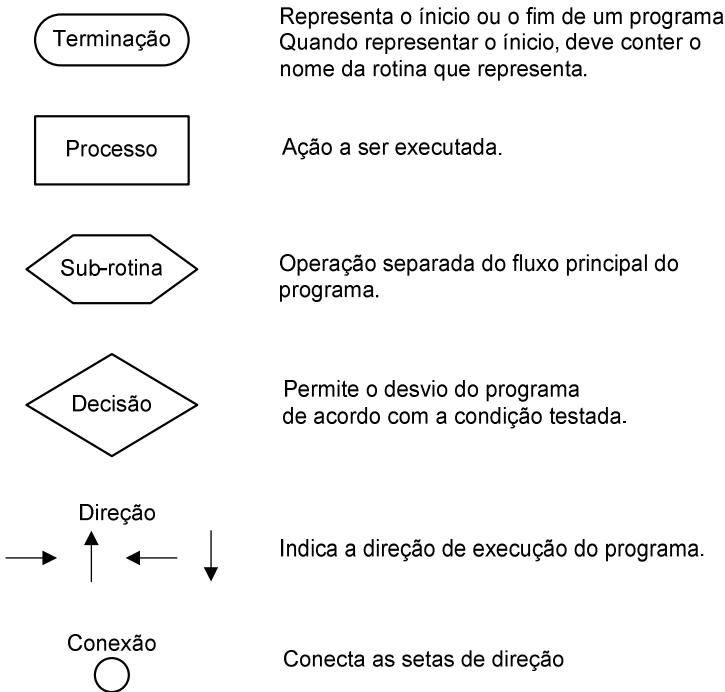


Fig. 4.1– Símbolos básicos para o desenho de fluxogramas.

Para exemplificar, na fig. 4.2, é apresentado o fluxograma para o algoritmo de conversão de um número decimal em seus dígitos individuais. A barra (/) indica que o resultado da divisão será um número inteiro e o símbolo de percentagem (%) indica que o resultado da divisão será o resto.

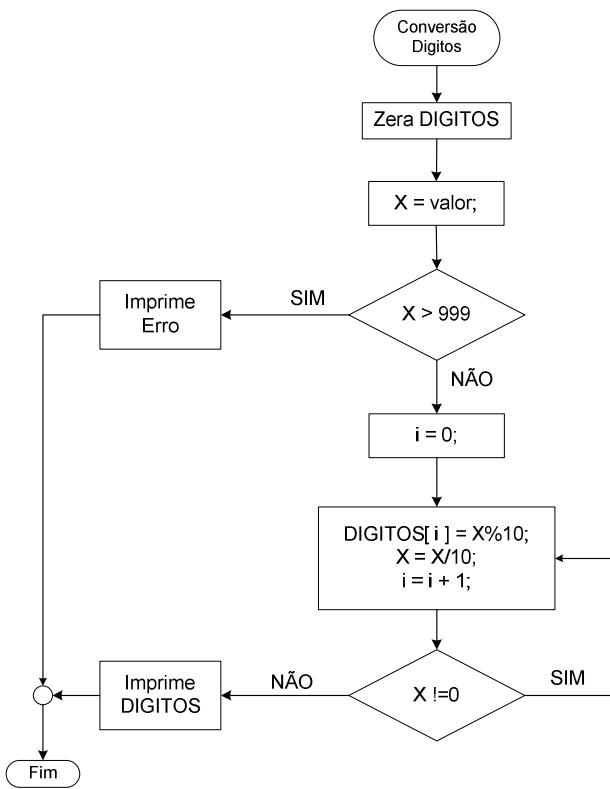


Fig. 4.2– Fluxograma para o algoritmo de cômputo dos dígitos individuais de um número decimal.

4.3 MÁQUINA DE ESTADOS

Outra maneira de representar um processo, além dos fluxogramas, é através da máquina de estados. Essa é muito comum em projetos de eletrônica digital, porém não muito usual em programação. Entretanto, é uma ferramenta interessante e alguns programadores gostam de utilizá-la. Uma máquina de estados é definida por um conjunto de ações que ocorrem sequencialmente em estados bem definidos. Na fig. 4.3, é representada uma máquina de estados para o algoritmo do cômputo dos dígitos individuais para um número decimal, apresentado anteriormente.

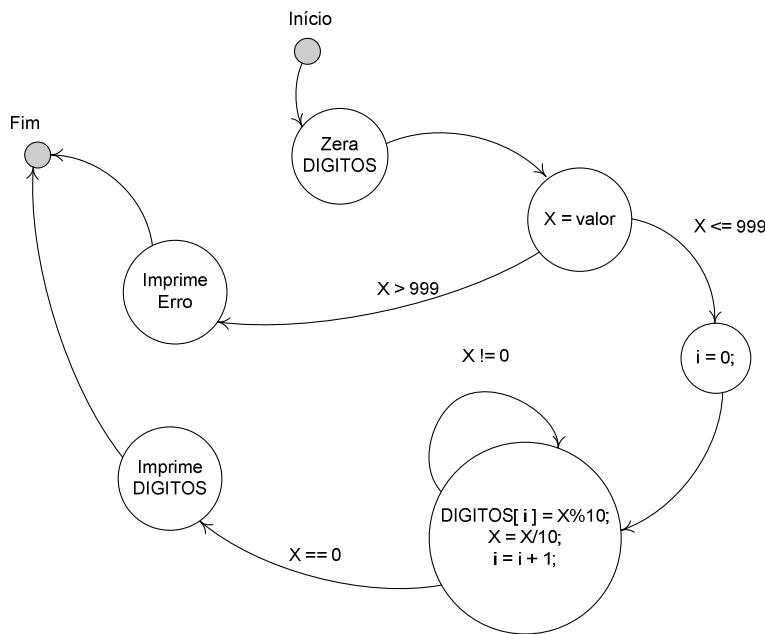


Fig. 4.3– Máquina de estados para o algoritmo de cômputo dos dígitos individuais de um número decimal.

4.4 O ASSEMBLY

Todo microprocessador¹⁵ possui um conjunto próprio de instruções, as quais são decodificadas por um circuito próprio. Essas instruções são representadas por mnemônicos que após o desenvolvimento do programa são convertidos nos zeros e uns lógicos interpretáveis pelo microprocessador. Esses zeros e uns são literalmente denominados linguagem de máquina, embora o nome *assembly* costume ser indevidamente empregado como sinônimo.

O *assembly* é uma linguagem de baixo nível, ou seja, de maior complexidade de programação e a mais próxima do que é compreensível por um microprocessador. O *assembly* permite obter o máximo

¹⁵ Lembrar que todo microcontrolador é um microprocessador, ver capítulo 1.

desempenho de um microprocessador, gerando o menor número de bytes de programa combinados a uma maior velocidade de processamento. Todavia, o *assembly* só será eficiente se o programa estiver bem estruturado e empregar algoritmos adequados.

Programar em *assembly* exige muito esforço de programação, o que torna trabalhoso o desenvolvimento de programas. Entretanto, é importante para a compreensão das funcionalidades e características do microprocessador empregado e para otimização de trechos de código.

Para exemplificar o uso do *assembly*, a seguir é apresentado um programa para o ATmega328 feito no AVR Studio, que serve para piscar um LED ligado ao pino PB5 (trocar o estado do pino com certa frequência). Os registradores de trabalho empregados, R16:R19, são de 8 bits. Notar que quando um deles possui o valor zero¹⁶ e é decrementado em uma unidade, ele passa a ter o valor 255. O tempo entre o ligar e o desligar do LED é feito decrementando várias vezes os registradores R17:R19. Maiores detalhes sobre a programação *assembly* serão vistos no capítulo 5 (os significados das instruções encontram-se no apêndice A).

```
//PROGRAMA PARA PISCAR UM LED LIGADO AO PINO PB5 DO ATMega328
.equ LED      = PB5      //LED é o substituto de PB0 na programação
.ORG 0x000      //endereço na memória flash de início de escrita do código

INICIO:
    LDI R16,0xFF      //carrega R16 com o valor 0xFF
    OUT DDRB,R16      //configura todos os pinos do PORTB como saída

PRINCIPAL:
    SBI PORTB,LED    //coloca o pino PB0 em 5 V
    RCALL ATRASO     //chama a sub-rotina de atraso
    CBI PORTB,LED    //coloca o pino PB0 em 0 V
    RCALL ATRASO     //chama a sub-rotina de atraso
    RJMP PRINCIPAL   //vai para PRINCIPAL
```

¹⁶ Após a inicialização do microcontrolador, os valores dos registradores de trabalho R0:R31 são desconhecidos. Aqui, foram considerados zero para a simplificação do programa. Se necessário, o programador deve inicializá-los.

```

ATRASO:           //sub-rotina de atraso
    LDI R19,0x02 //carrega R19 com o valor 0x02
volta:
    DEC R17      //decrementa R17, começa com 0x00
    BRNE volta   //enquanto R17 > 0 fica decrementando R17
    DEC R18      //decrementa R18, começa com 0x00
    BRNE volta   //enquanto R18 > 0 volta a decrementar R18
    DEC R19      //decrementa R19
    BRNE volta   //enquanto R19 > 0 vai para volta
    RET          //retorna da sub-rotina

```

Para melhorar a compreensão do programa acima e, ainda, reforçar os conceitos de fluxograma e máquina de estados, abaixo são apresentadas as figs. 4.4 e 4.5, desenvolvidas previamente à programação.

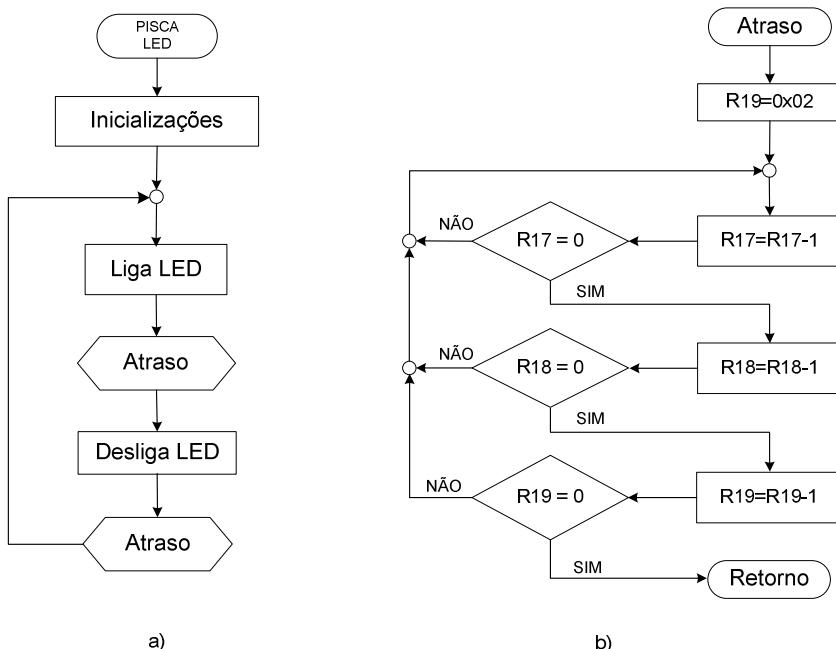


Fig. 4.4 – Fluxograma para o programa que pisca um LED: a) principal e b) sub-rotina.

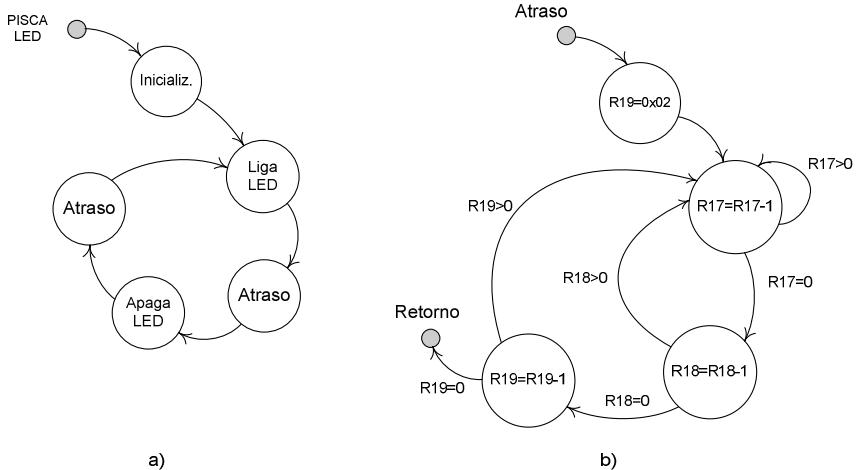


Fig. 4.5 – Máquina de estados para o programa que pisca um LED: a) principal e b) sub-rotina.

4.5 PROGRAMAÇÃO C

O domínio da linguagem *assembly* era total nos primórdios da programação de microcontroladores. Como o *assembly* é uma linguagem com baixo nível de abstração, o tempo gasto e o esforço mental são consideráveis no desenvolvimento de programas. Com a evolução tecnológica, o *assembly* foi quase que totalmente substituído pela linguagem C. Desta forma, os compiladores tem um papel importante, devendo eficientemente converter os programas em C para a linguagem *assembly* (compilação) e, posteriormente, para a de máquina (montagem).

As vantagens do uso do C comparado ao *assembly* são numerosas, tais como:

- Redução do tempo de desenvolvimento. Como o C é uma linguagem estruturada, fica relativamente fácil criar estruturas lógicas e escrever o código. Existe uma infinidade de bibliotecas com funções disponíveis. Não há necessidade do conhecimento profundo da arquitetura do microcontrolador a ser programado,

muito menos dos mnemônicos de programação *assembly*. A alocação de memória é feita automaticamente, sem a necessidade do conhecimento de registradores específicos.

- Como os códigos são de fácil escrita, o reuso dos códigos é facilitado, principalmente pela criação de funções próprias.
- Facilidade de manutenção. Pela estruturação da linguagem, corrigir problemas e alterar o código é relativamente simples.
- Portabilidade. Transferir o código de um tipo de microcontrolador para outro, mesmo com arquiteturas totalmente diferentes, é relativamente simples, fato complicado de ser realizado em *assembly*. A portabilidade é uma característica importantíssima, pois permite a migração rápida de uma tecnologia para outra.

O problema de desenvolver códigos em C é que os mesmos podem consumir muita memória e reduzir a velocidade de processamento. Os compiladores tentam traduzir da melhor forma o código para o *assembly* (antes de se tornarem código de máquina), mas esse processo não consegue o mesmo desempenho de um código escrito exclusivamente em *assembly*.

Como os compiladores C são eficientes para arquitetura do AVR, a programação dos microcontroladores ATmega é feita em C. Só existe a necessidade de se programar puramente em *assembly* em casos extremos. O bom é que os compiladores C aceitam trechos de código em *assembly* quando esse for imprescindível e vice-versa (ver o capítulo 20).

Além de um bom código em C, é necessário um bom compilador, pois, os mesmos têm influência direta na qualidade do código de máquina gerado. Desta forma, nada adianta um programa bem feito se o compilador for ruim.

4.5.1 INTRODUÇÃO

LINGUAGEM ESTRUTURADA

A linguagem C é estruturada, sua principal característica é a sequência de execução do programa; o programa inicia sua execução na primeira instrução e cada linha é executada, uma após a outra, até o final do programa. Também, é possível ignorar certas partes do programa, repetir outras partes ou até mesmo saltar de um ponto para outro. No entanto, a sequência lógica da execução do programa é de cima para baixo.

ENDENTAÇÃO

Endentação é o nome que se dá à técnica de separar graficamente cada bloco de instruções de acordo com seu grau de pertinência. Em algumas linguagens de programação antigas, a endentação era fundamental para a compilação do programa. Portanto, a correta endentação formava as estruturas corretamente.

Os compiladores da linguagem C não necessitam de endentação para funcionar corretamente: de fato, uma das primeiras ações que um compilador executa é ignorar toda a endentação e quebras de linha antes de compilar o programa. Essa medida facilitou a técnica de programação, mas gerou outro problema: possibilitou que os programadores escrevessem códigos desorganizados. Para demonstrar a utilidade da endentação, abaixo estão representados dois trechos de um programa:

```
for(;;)
{
if(!tst_bit(PINB,BOTAO))
{
if(valor==0xF)
valor=0;
else
valor++;
DISPLAY = Tabela[valor];
_delay_ms(250);
}}
```

```

for(;;)
{
    if(!tst_bit(PINB,BOTAO))
    {
        if(valor == 0xF)
            valor = 0;
        else
            valor++;

        DISPLAY = Tabela[valor];
        _delay_ms(250);
    }
}

```

Os dois códigos são funcionalmente idênticos, porém o segundo foi corretamente endentado e espaçado, tornando fácil a sua interpretação. Não é obrigatório endentar o código, no entanto, é fortemente aconselhado que o código seja endentado e bem organizado.

SENSIBILIDADE À CAIXA

Uma das mais importantes características da linguagem C é a sensibilidade à caixa, isto é, nomes de funções, variáveis e constantes declaradas são tratados como entidades diferentes se escritas com letras maiúsculas ou minúsculas.

```

int media; //uma variável
int Media; //outra variável
int MEDIA; //uma terceira variável
int MeDiA; //variável completamente diferente

```

Para evitar enganos devido à caixa, recomenda-se estipular um padrão de nomenclatura de variáveis, constantes e funções.

COMANDOS E PALAVRAS RESERVADAS

A linguagem C¹⁷ possui 32 comandos e palavras reservadas, ou seja, palavras que executam funções específicas e que não podem ser utilizadas como nomes de funções ou variáveis, conforme tab. 4.1.

¹⁷ O número de palavras reservadas do padrão ANSI C (C89)/C90 foi ampliada na versão ISO C (C99) com o acréscimo de cinco palavras: **_Bool**, **_Imaginary**, **_Complex**, **inline** e **restrict**. Como o padrão C99 não é completamente implementado em todas as versões de compiladores, o C90 será o padrão abordado neste livro.

Tab. 4.1. Palavras reservadas da linguagem C.

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

COMENTÁRIOS

Comentar um programa é extremamente importante para o entendimento correto de um programa feito há algum tempo. Pode-se dizer que é praticamente impossível entender um programa que não foi corretamente endentado e comentado. Portanto, aconselha-se que, sempre que possível, o programa seja comentado. Um bom comentário é o que traz informações relevantes sobre o código. O comentário demonstrado abaixo é inútil e deve se evitado, pois qualquer programador em linguagem C entenderia o trecho de código se o comentário fosse omitido.

```
double pi; //declara a variável pi
pi = 3.14; //atribui o valor do pi para a variável pi
```

Existem dois tipos de comentários: o comentário de linha simples e o comentário de múltiplas linhas. Os comentários de linha simples fazem com que o compilador ignore o trecho de código do início da instrução do comentário até o final da linha, enquanto que os comentários de múltiplas linhas instruem o compilador a ignorar o trecho de código entre a marcação do início do comentário até a marcação do final do comentário.

```
// Este é um comentário de apenas uma linha

/* Este é um comentário de múltiplas linhas,
o que permite escrever grandes quantidades
de texto, podendo-se explicar trechos complicados
do código ou especificar coisas que faltam ser
feitas */
```

VARIÁVEIS

Variável é um espaço na memória RAM, alocado para armazenar um determinado valor para uso futuro. Esse valor será modificado e utilizado para efetuar operações, comparações e outras instruções.

Antes de ser utilizada, uma variável deve ser declarada, ou seja, todas as variáveis utilizadas em uma função precisam ser relacionadas no início desta função. A declaração de variáveis é feita conforme exemplo a seguir.

```
<modificador de escopo> <modificador de tipo> <tipo> nome_da_variavel;
```

Em primeiro lugar, escreve-se o escopo, depois o tipo da variável, em seguida o nome da variável e termina-se a declaração com um ponto-e-vírgula (;).

Nome da Variável

O nome da variável deve seguir algumas exigências, tais como:

- O nome não pode conter outros caracteres além de letras, números ou o caractere sublinha (_).
- O primeiro caractere do nome deve ser uma letra ou o caractere de sublinha.
- Somente os primeiros 32 caracteres do nome são lidos. Dependendo do compilador utilizado, esse valor pode ser menor ainda.
- Letras maiúsculas e minúsculas são consideradas como diferentes, ou seja, o compilador é sensível à caixa da letra.

Tipo da Variável

O tipo da variável determina o espaço que ela ocupará na memória. O espaço na memória, por consequência, está ligado ao tamanho do dado que se deseja armazenar. Na linguagem C, existem cinco tipos de dados: **char**, **int**, **float**, **double** e **void**, conforme tab. 4.2. O tipo de dado **void** não é uma variável e será abordado na seção sobre funções.

Tab. 4.2. Tipos de variáveis.

Declaração	Bytes	Variação
char	1	-128 a 127
int *	2	-32.768 a 32.767
float	4	$1,2 \times 10^{-38}$ a $3,4 \times 10^{+38}$
double	8	$2,2 \times 10^{-308}$ a $1,8 \times 10^{+308}$

* Depende da arquitetura do processador, sistema operacional e compilador utilizado.

Modificador de Tipo

Os quatro tipos de dados podem receber os modificadores opcionais de modo a criar um tipo de dado mais adequado para cada aplicação. A instrução **short** informa ao compilador que o tamanho da variável necessária é menor que o padrão do tipo de dado; de forma análoga, a instrução **long** informa que o tamanho necessário é maior que o tamanho padrão do tipo de dado, e as instruções **unsigned** e **signed** informam que os valores armazenados serão apenas positivos ou positivos e negativos, respectivamente (ver a tab. 4.3).

Tab. 4.3 Modificadores de tipo.

Declaração	Bytes	Variação
signed char	1	-128 a 127
unsigned char	1	0 a 255
short	2	-32.768 a 32.767
short int	2	-32.768 a 32.767
signed int	2	-32.768 a 32.767
signed short	2	-32.768 a 32.767
signed short int	2	-32.768 a 32.767
unsigned int	2	0 a 65.535
unsigned short	2	0 a 65.535
unsigned short int	2	0 a 65.535
long	4	-2.147.438.648 a 2.147.438.647
long int	4	-2.147.438.648 a 2.147.438.647
signed long	4	-2.147.438.648 a 2.147.438.647
signed long int	4	-2.147.438.648 a 2.147.438.647
unsigned long	4	0 a 4.294.967.295
unsigned long int	4	0 a 4.294.967.295
long double*	8	$2,2 \times 10^{-308}$ a $1,8 \times 10^{+308}$

* Dependendo do compilador pode ser atribuído 8, 10 ou 12 bytes.

Se, por exemplo, deseja-se armazenar a idade de uma pessoa, pode-se utilizar um dado do tipo **char**, por armazenar valores entre -128 a 127, ou um dado do tipo **unsigned char**, por se tratar de um dado apenas positivo. Nesse caso, a escolha fica a cargo do programador. Pode-se perceber que vários tipos de dados são sinônimos. Isso acontece, por exemplo, com todos os dados **signed**. O padrão de qualquer variável criada é ser sinalizada e, portanto, o modificador **signed** normalmente é suprimido.

Conversão de Tipo

É fundamental que o tipo de variável declarada no programa seja adequada às operações que serão realizadas com ela. O erro mais comum quando se aprende a linguagem C é atribuir um tipo errado às variáveis, por exemplo:

```
unsigned char a, b, resultado;  
...  
resultado = a + b;
```

Como o resultado foi declarado com **unsigned char** (1 byte de tamanho) o número máximo que ele pode representar é 255. Assim, a soma acima pode estourar sua capacidade, gerando um resultado errado. O certo seria declarar resultado como um **unsigned int** (2 bytes de tamanho).

Muitas vezes se realizam operações com variáveis de tipos diferentes. Nesse caso o C faz a conversão automática; pode-se dizer que a variável menor sempre caberá em uma maior e o inverso gerará um truncamento. Entretanto, deve-se ter cuidado quando a conversão se dá com variáveis que não são **int**, **signed** ou **float**. Nesse caso, pode ser realizada uma conversão forçada (*casting*), por exemplo:

```
float a = 6.7;  
int b;  
...  
b = (int)a; //b terá o valor 6.
```

Modificador de Escopo

O modificador de escopo é um parâmetro opcional que define qual a área de existência da variável. Quando se declara uma variável ela existe localmente, isto é, dentro da função em que foi declarada. As variáveis locais são chamadas de automáticas (**auto**) e pode-se omitir esse modificador.

As variáveis podem ser declaradas também dentro dos registradores através do modificador **register**. O comando solicita ao microcontrolador que a variável seja armazenada em um registrador interno. As variáveis nos registradores são acessadas em um tempo menor, pois o acesso é mais rápido que o da memória RAM. Os registradores não podem ser utilizados em variáveis globais, pois isso resultaria num registrador ocupado todo o tempo por essa variável. O modificador **register** é uma solicitação e, portanto, não necessariamente será atendido. Atualmente, a maioria dos compiladores para microcontroladores (inclusive os do AVR), quando possível, usam registradores para as variáveis.

As variáveis marcadas como **const** não podem ter seu valor modificado, de modo que qualquer tentativa de modificar seu valor através de atribuição, incremento ou decremento resulta num erro de compilação.

O modificador **volatile** define uma variável que pode ser modificada sem o conhecimento do programa principal, mesmo que essa ainda esteja declarada dentro do escopo onde o programa está sendo executado. Portanto, o compilador não pode prever com segurança se é possível otimizar trechos de programa onde essa variável se encontra. Esse modificador é destinado às situações nas quais uma variável pode ter seu valor alterado por fatores diversos e o compilador não deve interferir na forma como o programa foi escrito para o acesso dessa variável. Isso impede que erros inseridos por otimização estejam presentes na versão final do programa executável.

O modificador **static**, quando utilizado em uma variável global, torna essa variável conhecida apenas dentro de um módulo (arquivo), e desconhecida em outros módulos (arquivos). Pode-se utilizar o modificador **static** para isolar trechos de um programa para evitar mudanças accidentais em variáveis globais. No caso de variáveis locais, o modificador **static** define que uma variável manterá seu valor entre as chamadas da função.

O modificador **extern** indica ao compilador que a variável foi declarada em outro módulo (arquivo) compilado separadamente. Os dois arquivos serão ‘unidos’ (*linker*) para formar o programa final. Se não for utilizado o modificador **extern**, o compilador irá declarar uma nova variável com o nome especificado, ocultando a variável que realmente se deseja usar.

OPERADORES

A linguagem C possui um grande número de operadores: lógicos, aritméticos, relacionais, bit a bit e outros. Esses operadores são empregados para definir as ações do programa. Eles são apresentados na tab. 4.4. A coluna de precedência indica a ordem de prioridade na execução de uma ação, por exemplo, supondo a operação $3 \times 5 + 2$, a multiplicação (precedência = 3) será executada primeiro que a soma (precedência = 2). Entretanto, a soma de dois números pode ser executada primeiro se estiver entre parênteses $3 \times (5 + 2)$, precedência do parêntese = 1. O programa vai executar a operação de acordo com a associatividade do operador, podendo começar a operação da esquerda para a direita ou vice-versa.

Tab. 4.4. Operadores da linguagem C.

Operador	Nome do Operador	Função	Precedência	Associatividade
()	Priorizador de operação.	Prioriza uma operação. Também utilizado em funções.	1	esq. p/ dir.
[]	Elemento de matriz	Indica um elemento de uma matriz		
->	Ponteiro para elemento de estrutura	Retorna um elemento da estrutura ou união.		
.	Membro de estrutura	Retorna um elemento da estrutura ou união.		
!	Negação lógica	Retorna verdadeiro caso o operando seja falso.	2	dir. p/ esq.
~	Complemento de um	Inverte todos os bits do operando.		
++	Incremento	Soma um ao operando.		
--	Decremento	Diminui um do operando.		
-	Menos unário	Retorna o operando negativo.		
*	Unário: ponteiro	Retorna o valor do operando que contém um endereço.		
&	Endereço (ponteiro)	Retorna o endereço do operando (variável ou nome de função).		
sizeof	Tamanho do objeto	Retorna o número de bytes do operando na memória do sistema.		
*	Multiplicação	Multiplica dois operandos.	3	esq. p/ dir.
/	Divisão	Divide dois operandos.		
%	Resto da divisão	Retorna o resto da divisão entre dois operandos inteiros.		
+	Adição	Adiciona dois operandos.	4	esq. p/ dir.
-	Subtração	Subtrai dois operandos.		
<<	Deslocamento à esquerda	Desloca os bits do operando x bits à esquerda.	5	esq. p/ dir.
>>	Deslocamento à direita	Desloca os bits do operando x bits à direita.		
<	Menor que	Retorna verdadeiro se o operando da esquerda for menor que o operando da direita.	6	esq. p/ dir.
<=	Menor ou igual a	Retorna verdadeiro se o operando da esquerda for menor ou igual ao operando da direita.		
>	Maior que	Retorna verdadeiro se o operando da esquerda for maior que o operando da direita.		
>=	Maior ou igual a	Retorna verdadeiro se o operando da esquerda for maior ou igual ao operando da direita.		

Operador	Nome do Operador	Função	Precedência	Associatividade.
==	Igualdade	Retorna verdadeiro se o operando da esquerda e da direita forem iguais.	7	esq. p/ dir.
!=	Desigualdade	Retorna verdadeiro se o operando da esquerda e da direita forem diferentes.		
&	AND, bit a bit	Lógica E entre os bits dos operandos.	8	esq. p/ dir.
^	XOR, bit a bit	Lógica OU EXCLUSIVO entre os bits dos operandos.	9	esq. p/ dir.
 	OR, bit a bit	Lógica OU entre os bits dos operandos.	10	esq. p/ dir.
&&	AND lógico	Retorna verdadeiro caso ambos os operandos sejam verdadeiros.	11	esq. p/ dir.
 	OR lógico	Retorna verdadeiro caso um dos operandos seja verdadeiro ou ambos.	12	esq. p/ dir.
?	Condicional	Avalia a primeira expressão, caso seja verdadeira, retorna o resultado da segunda expressão. Sendo falsa retorna o resultado da terceira.	13	dir. p/ esq.
=	Atribuição	O operando da esquerda recebe o valor do operando da direita ou o resultado de uma expressão.	14	dir. p/ esq.
,	Vírgula (para encadeamento de expressões)	Realiza uma sequência de operações dentro de uma atribuição	15	esq. p/ dir.

Escrevendo operações

A linguagem C permite uma escrita compacta de várias operações, alguns exemplos são apresentados na tab. 4.5.

Tab. 4.5. Escrita de operações.

Forma compacta	Forma normal
$a += b$	$a = a + b$
$a -= b$	$a = a - b$
$a *= b$	$a = a * b$
$a /= b$	$a = a / b$
$a %= b$	$a = a \% b$
$a &= b$	$a = a \& b$

Forma compacta	Forma normal
a = b	a = a b
a ^= b	a = a ^ b
a <= b	a = a << b
a >= b	a = a >> b

Operadores Pré e Pós-fixados

Uma peculiaridade do C é a forma de incrementar e decrementar uma variável, permitindo gerar um código mais eficiente, para tal são utilizados os operadores `++` e `--`. Por exemplo:

```
a = a + 1;    =>      a++;  ou      ++a;
b = b - 1;    =>      b--;  ou      --b;
```

Quando os operadores precedem o operando, o incremento ou decremento é feito antes de se empregar o mesmo, se o sucedem, o incremento é feito após o seu uso. Isso é válido quando se empregam mais de um operando, por exemplo:

```
a = 2;
b = ++a;
```

O que resulta em `b = 3`. Se `b = a++`, o resultado seria `b = 2` (`a` será 3, mas o incremento se dá após o seu emprego).

CORPO DE UM PROGRAMA

Um programa em C é composto basicamente por um conjunto inicial de definições, funções e o corpo principal do programa. Assim, um programa constará das seguintes partes:

- 1º inclusão de arquivos externos (bibliotecas de funções, declarações, dados).
- 2º definições de nomes e macros (para facilitar a escrita do código).
- 3º declaração de variáveis globais (disponíveis para qualquer parte do programa).
- 4º declaração de protótipos de funções (declaração das funções desenvolvidas).
- 5º programa principal - **main()** (pode chamar uma ou mais funções).
- 6º funções (podem ser escritas abaixo do **main()** caso declaradas antes dele).

Existem inúmeras bibliotecas com funções prontas disponíveis ao programador, bastando apenas incluí-las no início do programa. O compilador AVR-GCC, para uso com os microcontroladores AVR, disponibiliza uma infinidade de bibliotecas para facilitar a programação.

No inicio do programa são definidas as variáveis que serão empregadas, constantes e outras declarações, como por exemplo, as funções desenvolvidas no corpo do programa.

A função main()

A função **main()** é a função principal do programa. Quando o programa é executado, o controle do microcontrolador é passado ao programa do usuário que começa a executar a primeira linha da função **main()**. Normalmente, em um microcontrolador, deseja-se que o programa seja executado ininterruptamente. Assim, faz-se necessária a utilização de um laço infinito. Dessa forma, o menor programa possível em C pode ser escrito como:

```
void main()
{
    while(1)
    {
        ...
    }
}
```

4.5.2 ESTRUTURAS CONDICIONAIS

Estruturas condicionais são estruturas que permitem a execução condicional de uma ou mais instruções.

if – else (se – senão)

A estrutura **if – else** é a estrutura de controle de fluxo mais utilizada em programação. Ela pode ser implementada de forma simples, composta e até de forma aninhada.

Quando implementada de forma simples, um **if** é uma verificação seguida de uma execução se a condição for verdadeira. Na implementação composta, além do **if**, emprega-se uma instrução **else**. Se a condição determinada pelo **if** for falsa, as instruções delimitadas pelo **else** serão executadas.

O comando **if** apresenta uma morfologia bem definida. Se o bloco de instruções possuir apenas um comando, os delimitadores de bloco (chaves) podem ser suprimidos.

```
if(<condição>
{
    <instruções>
}
else
{
    <instruções>
}
```

As condições também podem ser agrupadas dentro de uma mesma estrutura **if** através dos operadores relacionais **&&**, **||** e **!**. Também é possível aninhar **ifs** e **elses**, ou seja, criar uma estrutura onde várias condições diferentes são testadas e as instruções são executadas de acordo com as condições estabelecidas.

```

if(<condição 1>
{
    if(<condição 2>
    {
        <instruções>
    }
    else
    {
        <instruções>
    }
}
else
{
    if(<condição 3>
    {
        <instruções>
    }
    else
    {
        <instruções>
    }
}

```

Embora pareça mais confusa, a estrutura de **ifs** e **elses** aninhados é mais inteligente que uma sequência grande de **ifs** ao longo do programa. Isso porque em programas grandes, o processador efetua menos verificações, tornando o algoritmo mais veloz.

O operador ternário condicional (**? :**) é uma alternativa ao uso da estrutura **if - else** quando essa utiliza expressões simples. O operador ternário é assim chamado porque utiliza três termos como parâmetros. Considere a seguinte implementação **if - else**, onde a variável **max** recebe o maior valor entre **x** e **y**:

```

if (x > y)
    max = x;
else
    max = y;

```

Utilizando o operador ternário condicional, obtém-se uma implementação mais compacta:

```
max = x > y ? x : y;
```

switch - case (chave - caso)

Pode-se utilizar **switch** nos casos em que uma variável pode assumir vários valores. Desta forma, descrevem-se ações para cada estado previsto da variável e ações padrão para todos os outros casos. Essa estrutura é muito útil quando se necessita de muitos **ifs** e **elses** para a delimitação de condições. No entanto, não é recomendável para verificações simples. Sua estrutura é dada por:

```
switch(<variável>
{
    case <condição 1>:   <instruções>
        break;
    case <condição 2>:   <instruções>
        break;
    ...
    case <condição n>:   <instruções>
        break;
    default:             <instruções padrão>
        break;
}
```

Observa-se que a estrutura **switch** não substitui totalmente as cláusulas **ifs** e **elses**, pois não há como relacionar mais de uma variável ao mesmo tempo. Ela relaciona todos os estados possíveis de apenas uma variável. Também é importante verificar que a instrução **break** aparece sempre após o final de um bloco **case**. Essa instrução se faz necessária para que o próximo bloco **case** não seja executado.

4.5.3 ESTRUTURAS DE REPETIÇÃO

As estruturas de repetição são instruções que devem ser realizadas enquanto uma determinada condição for verdadeira.

A instrução **break** é chamada instrução de quebra, pois quando ela é executada, ela aponta para a próxima instrução fora do laço de repetição corrente. Ou seja, ela informa ao programa que o laço de repetição deve ser ignorado e não deve mais ser processado, passando o controle para a próxima instrução após o laço.

A instrução **continue**, por outro lado, informa que o laço de repetição atual deve ser ignorado e que um novo laço deve ser executado.

while (enquanto)

A estrutura **while** é a estrutura de repetição mais simples de ser implementada. Sua sintaxe é bastante elementar e através dela é possível gerar, rapidamente, laços (*loops*) infinitos e finitos, com bastante facilidade.

```
while(<condição>)
{
    <instruções>
}
```

Assim, um laço infinito pode ser feito colocando-se o número 1 (**true** - verdadeiro) no lugar da condição e, um contador, através da utilização da variável da condição sendo incrementada ou decrementada nas instruções.

```
while(1)                  //repete infinitamente
{
    <instruções>
}

int numero = 0;
while(numero < 10)      //repete 10 vezes
{
    <instruções>
    numero++;
}
```

Se o bloco de instruções da estrutura **while** apresentar apenas uma instrução, os delimitadores de bloco não são necessários. Normalmente uma estrutura **while** aparece com estruturas condicionais mescladas às suas instruções, permitindo um controle mais efetivo do laço de repetição.

do – while (faça – enquanto)

A estrutura **do while** funciona da mesma forma que a estrutura **while**, exceto pelo fato das instruções serem executadas primeiro e depois a verificação da condição.

```
do
{
    <instruções>
}while(<condição>); //é necessário o ponto e vírgula aqui
```

for (para)

A estrutura **for** é a estrutura de repetição mais difundida.

```
for(<condição inicial>;<condição de parada>;<incremento de passo>)
{
    <instruções>
}
```

Elá realiza os seguintes procedimentos: atribui a condição inicial, verifica a condição de parada, executa as instruções e incrementa o passo. Nos laços seguintes ao primeiro, a atribuição inicial não é mais executada.

Pode-se omitir cada um dos argumentos do **for** de modo a criar efeitos interessantes, como por exemplo:

- Omitindo-se a condição inicial, pode-se utilizar o valor de uma variável pré-definida como valor inicial do laço **for**.

```
int numero = 14;
int x;
for(x = 0;x < numero;x++) //x entre 0 e número
{
    <instruções>
}
for( ;x < 50;x++)           //x entre número e 50
{
    <instruções>
}
```

- Omitindo-se a condição de parada, pode-se criar um laço infinito ou um laço condicional.

```
int numero = 14;
int x;
for(x = 0; ;x++)      //laço condicional
{
    if(numero == x)
        break;
    <instruções>
}
```

- Omitindo-se o incremento de passo, também se pode criar um laço infinito ou um laço condicional.

```
int numero = 14;
int x;
for( ;x < numero; )  //laço condicional
{
    if(numero == x)
        break;
    <instruções para modificar x>
}
```

Outra opção para criar um laço infinito é a omissão dos três argumentos, ou através de combinações especiais de alguns argumentos. Enfim, diversas sintaxes podem ser utilizadas.

É importante salientar que tanto o **while** quanto o **for** podem ser utilizados em todas as situações possíveis, não existindo casos em que um possa ser implementado enquanto o outro não possa. Desta forma, os laços **while** e **for** não são complementares, eles são análogos; cabe ao programador decidir qual dos dois utilizar, se valendo da facilidade de implementação como critério de seleção para cada caso.

DESVIOS E RÓTULOS

Uma forma bem interessante e controversa de controle do fluxo de execução do programa é o desvio. Ele é realizado com a instrução **goto** que remete para um rótulo.

Os desvios são estruturas polêmicas, pois sua utilização pode causar um funcionamento imprevisível para o código e erros difíceis de serem detectados. Algumas vezes desvios mal implementados podem gerar laços infinitos indesejados e a sua utilização é, de maneira não formal, evitada.

```
<instruções 1>
repetir:           //rótulo
    <instruções 2>
    if(<condição>)
        goto repetir;   //salto
```

O desvio, quando utilizado para acessar rótulos anteriores a ele, realiza a função da estrutura **while** ou **for**, podendo-se criar laços infinitos, finitos, condicionais ou quaisquer outras funcionalidades dessas estruturas. Quando acessa rótulos posteriores a ele, realiza a função da estrutura **if-else** ou **switch-case**, determinando trechos de código que não serão executados.

4.5.4 DIRETIVAS DE PRÉ-COMPILAÇÃO

Conceitualmente, a linguagem C é baseada em blocos de construção. Assim sendo, um programa em C nada mais é que um conjunto de funções básicas ordenadas pelo programador. Algumas instruções não fazem parte do conjunto de palavras padrão da linguagem C. Essas palavras desconhecidas são blocos de instruções ordenados de forma a realizar um determinado processamento.

Bibliotecas e cabeçalhos

Todos os códigos-fonte na linguagem C necessitam de arquivos de bibliotecas e cabeçalhos para serem compilados. Os arquivos de cabeçalho (*headers*) são pequenos arquivos contendo bits de código de programa, os quais são utilizados para referenciar e descrever o conteúdo de outros módulos, as bibliotecas (*libraries*).

Esta subdivisão de arquivos é feita para facilitar o trabalho do programador de bibliotecas, pois o arquivo de cabeçalho contém as informações que mudam constantemente a cada atualização da biblioteca, como por exemplo, informação de interfaces, enquanto que o arquivo da biblioteca contém apenas as modificações estruturais.

Nesse caso, os arquivos de cabeçalho contêm apenas as definições das funções para cada interface, juntamente com parte da documentação. O sistema necessita somente verificar a data do arquivo de cabeçalho para saber se é necessária a recompilação do código-fonte ou se houve alguma mudança significativa na biblioteca. No entanto, quando se realiza uma modificação na biblioteca, também é necessário modificar o arquivo de cabeçalho.

Diretiva #include

A diretiva **#include** é certamente a diretiva de pré-compilação mais usada na linguagem C. Essa diretiva copia o conteúdo de um arquivo àquele ponto do código-fonte. As inclusões mais comuns são arquivos de cabeçalho de funções de entrada e saída de dados.

Por um lado, esta abordagem torna o reuso de código simples, porém pode acarretar problemas de múltiplas inclusões de um mesmo arquivo. Para prevenir que um mesmo arquivo seja incluído mais de uma vez, gerando um erro de compilação, pode-se utilizar um **include guard**, um trecho de código que testa se o arquivo já foi incluído.

A morfologia da diretiva **include** é a seguinte:

```
#include <nome_do_arquivo.extensão>
#include "nome_do_arquivo.extensão"
```

A diferença entre essas duas formas da diretiva é que, quando o pré-processador encontra uma declaração com o nome do arquivo envolto por `<` e `>`, o pré-processador procura por um cabeçalho específico nos diretórios de cabeçalhos e o processa, incluindo apenas a parte que realmente foi utilizada. Por sua vez, quando a diretiva possui o nome do arquivo envolto por `"` e `"`, o pré-processador procura o cabeçalho no diretório de trabalho atual.

Diretiva #define

Se a diretiva **#include** é a mais usada na linguagem C, a diretiva **#define** é a segunda. Esta diretiva possui três funções bem definidas: define um símbolo para ser testado mais tarde, define uma constante que será utilizada no programa e define uma macro e seus parâmetros.

```
#define nome_do_simbolo  
#define nome_da_constante seu_valor  
#define nome_da_macro(parâmetros) expressao_equivalente
```

Quando o pré-compilador encontra a constante **nome_da_constante**, ele a substitui por **seu_valor** e quando ele encontra a macro **nome_da_macro**, ele a substitui pela **expressao_equivalente**.

Diretiva #undef

A diretiva **#undef** apaga o símbolo, a constante ou a macro da tabela de substituição e o pré-compilador passa a não reconhecer mais aquela sequência.

```
#undef nome_do_simbolo  
#undef nome_da_constante  
#undef nome_da_macro
```

Diretivas #ifdef, #ifndef, #if, #endif, #else, #elif

Estas diretivas são utilizadas como estruturas condicionais. As diretivas **#if** e **#endif** funcionam como a estrutura **if** da linguagem C.

```
#if expressão  
    código  
#endif
```

As diretivas **#if** e **#endif** podem conter um bloco **#else**, para funcionar como a estrutura **if-else** da linguagem C.

```
#if expressão  
    código 1  
#else  
    código 2  
#endif
```

As diretivas **#if** e **#endif** podem conter blocos **#elif**, para funcionar como a estrutura **if-else** aninhados da linguagem C. O uso de um bloco **#else** ao final também é possível.

```
#if expressão 1
    código 1
#elif expressão 2
    código 2
#elif expressão 3
    código 3
#else
    código final
#endif
```

As diretivas **#ifdef** e **#ifndef** podem ser usadas para implementar código de forma condicional à existência (**#ifdef**) ou não (**#ifndef**) de um símbolo, ambas finalizadas por **#endif**.

```
#ifdef símbolo
    código
#endif

#ifndef símbolo
    código
#endif
```

O uso mais comum das diretivas de compilação é em arquivos de cabeçalho que só devem ser incluídos uma vez. Muitas vezes um arquivo é incluído em diversos arquivos e esses arquivos podem ser incluídos em outros arquivos e, de forma não intencional, o cabeçalho é incluído diversas vezes. Para evitar problemas, costuma-se envolver o arquivo inteiro com um bloco condicional que só será compilado se o arquivo não tiver sido incluído. A esse bloco, dá-se o nome de **include guard**.

O **include guard** é implementado através da definição de um símbolo baseado normalmente no nome do arquivo.

```
// Arquivo cabecalho.h
#ifndef CABECALHO_H
#define CABECALHO_H
    código
#endif
```

Se o arquivo ainda não tiver sido incluído, ao chegar à primeira linha do arquivo, o pré-processador não encontrará o símbolo CABECALHO_H, e continuará a ler o arquivo, o que lhe fará definir o símbolo. Se houver uma tentativa de incluir novamente o arquivo, o pré-processador pulará todo o conteúdo uma vez que o símbolo já foi definido.

4.5.5 PONTEIROS

Ponteiro é uma variável que contém um endereço de memória, como o próprio nome diz, ele aponta indiretamente para uma variável armazenada no seu endereço. A grande vantagem do uso de ponteiros é o aumento da eficiência do código produzido. A declaração de um ponteiro é feita empregando-se o operador asterisco (*), da seguinte forma:

```
tipo *nome_ponteiro;
```

Por exemplo: `unsigned char *dados;`

Para a leitura do endereço apontado emprega-se o operador &. Assim:

```
a = *dados; //a tem o valor do conteúdo dados apontado pelo ponteiro.  
b = &dados; //b tem o valor do endereço da variável dados.
```

Na programação de microcontroladores, os ponteiros são utilizados para a varredura de tabelas de dados, principalmente com caracteres. Em *assembly* fica mais fácil entender o conceito de ponteiros (ver a seção 5.4 para uma tabela de dados).

4.5.6 VETORES E MATRIZES

Os vetores ou matrizes são variáveis com mais de uma posição de memória e do mesmo tipo, permitindo a criação de variáveis com uma ou mais dimensões. A seguir são apresentados alguns exemplos.

Vetor (uma dimensão):

```
char teste[10]; // aloca 10 posições de memória char  
char teste[ ]= "Uma string"; // inicializa a variável de uma só vez  
char teste[10]={'U', 'm', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g'};  
                // inicializa a variável individualmente por posição.
```

Um vetor tem somente uma dimensão (um vetor é uma matriz de uma única dimensão) e pode ser inicializado, como nos exemplos acima. Quando se escreve uma cadeia de caracteres (*string*) e não é informado o tamanho do vetor, o compilador C irá colocar automaticamente um caractere nulo no final da *string*, alocando uma posição a mais de memória, isso se deve porque se costuma utilizar *strings* com ponteiros e o caractere nulo ajuda na identificação do final da *string*.

Matriz (n dimensões):

```
int teste[3][4]; // aloca 12 posições int da memória  
// inicialização da matriz  
int teste[3][4]={{0xADFC, 0x0001, 0x345B, 0x1029},  
                 {0xA88C, 0x0021, 0x0CB1, 0x1229},  
                 {0x980C, 0x1001, 0x3EDB, 0xAAAA}};
```

Uma matriz poderá conter **n** dimensões de acordo com o número de colchetes empregados []. No caso de mais de uma dimensão, é necessário indicar o número de posições de memória alocados. Quando se inicializa uma matriz deve-se ter cuidado na forma da sua escrita, as linhas com o respectivo número de colunas podem ser colocadas entre chaves {} para facilitar a visualização.

4.5.7 ESTRUTURAS

Estruturas são agrupamentos de variáveis de qualquer tipo e podem facilitar a programação.

```
struct nome_estrutura // nome_estrutura ou variaveis_estrutura pode  
{                                // ser omitido (um ou outro)  
    tipo variavel_1;  
    tipo variavel_2;  
    ...  
    tipo variavel_N;  
  
}variaveis_estrutura;
```

Por exemplo:

```
struct cadastro
{
    char nome[20];
    char cidade[15];
    unsigned char senha[3];
    unsigned long int CEP;

}dados[20];//20 variáveis dados com a estrutura cadastro
```

O acesso à variável se dá com o emprego do ponto (.). Por exemplo, a escrita de dados[0] se dá como:

```
dados[0].nome = "Fulano numero zero ";
dados[0].cidade = "Florianopolis ";
dados[0].senha[0]=2;
dados[0].senha[1]=8;
dados[0].senha[3]=1;
dados[0].CEP = 88000;
```

4.5.8 FUNÇÕES

As funções são muito importantes em programação, pois simplificam o código e evitam a repetição de comandos. Uma função em C é na verdade uma sub-rotina que contém um ou mais comandos em C e que executa uma ou mais tarefas. Em um programa bem escrito, cada função deve executar uma tarefa modular. Cada função deverá possuir um nome e a lista de argumentos que receberá. A forma geral de uma função é:

```
tipo_variavel_retorno    nome_funcao(variáveis de entrada)
{
    comandos;
    ...
}
```

Exemplos:

1- void atraso(int repet) //a função recebe o nr. de repetições
{
 for(unsigned int i=0; i<repet; i++); //executa o laço for
 //repet vezes
}

A função acima gera um atraso na execução do programa, repetindo o laço **for** pelo número de vezes do valor passado à função. A função não

retorna nenhum valor (**void** - vazio) apenas gasta um tempo de processamento.

2- float Fahr2Cels(float F) //converte uma valor de temperatura de Fahrenheit para graus Celsius
{
 //Fahrenheit para graus Celsius
 float C;

 C = (F-32)/1.8;
 return (C);
}

A função acima converte um valor de temperatura de Fahrenheit para graus Celsius. Ela recebe e retorna um número **float**. Um programa exemplo que poderia utilizar tal função seria:

```
float valor, temp;  
...  
//declaração das funções empregadas  
float leitura_sensor();  
float Fahr2Cels(float F);  
...  
void main()  
{  
    valor = leitura_sensor(); //Em algum lugar do programa existe uma  
                            //função que lê um determinado sensor de  
                            //temperatura e retorna o seu valor em  
                            //Fahrenheit.  
    temp = Fahr2Cels(valor); //Converte o valor lido para graus Celsius e  
                            //o escreve em temp.  
    ...  
}  
  
float leitura_sensor()  
{  
    //aqui vai o código para a leitura do sensor  
    return (temp_F);  
}  
float Fahr2Cels(float F)  
{  
    //aqui vai o código para a conversão de unidades  
    return (C);  
}
```

Uma função pode trabalhar com variáveis e não retornar nenhum número (**void**), basta que ela altere variáveis globais, disponíveis em qualquer parte do programa. Da mesma forma, funções podem receber e trabalhar com ponteiros (passagem por referência).

4.5.9 O IMPORTANTÍSSIMO TRABALHO COM BITS

O trabalho com bits é fundamental para a programação de um microcontrolador. Assim, compreender como podem ser realizadas operações com bits é primordial para uma programação eficiente.

Quando se programa em C, é comum utilizar números com notação decimal, hexadecimal ou binária e seu uso depende da conveniência. No compilador AVR-GCC, essas notações são dadas por:

DECIMAL - qualquer número no formato decimal (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

HEXADECIMAL - começa com 0x seguido pelo número hexadecimal (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

BINÁRIO - começa com 0b seguido por zeros ou uns (0, 1), o bit menos significativo (LSB) fica a direita e o mais significativo (MSB) a esquerda.

Na tab. 4.6, é apresentado a equivalência entre as notações. No apêndice F é dada uma tabela para todos os números entre 0 e 255.

Tab. 4.6. Equivalência entre números decimais, hexadecimais e binários.

Decimal	Hexadecimal		Binário			
			Bit 3 (MSB)	Bit 2	Bit 1	Bit 0 (LSB)
0	0x0	0b	0	0	0	0
1	0x1	0b	0	0	0	1
2	0x2	0b	0	0	1	0
3	0x3	0b	0	0	1	1
4	0x4	0b	0	1	0	0
5	0x5	0b	0	1	0	1
6	0x6	0b	0	1	1	0
7	0x7	0b	0	1	1	1
8	0x8	0b	1	0	0	0
9	0x9	0b	1	0	0	1
10	0xA	0b	1	0	1	0
11	0xB	0b	1	0	1	1
12	0xC	0b	1	1	0	0
13	0xD	0b	1	1	0	1
14	0xE	0b	1	1	1	0
15	0xF	0b	1	1	1	1

Como cada número hexadecimal corresponde a 4 bits, é fácil converter um número hexadecimal para binário e vice-versa, por exemplo:

0x	A				8			
0b	1	0	1	0	1	0	0	0
0x	F				0			
0b	1	1	1	1	0	0	0	0
0x	C				5			
0b	1	1	0	0	0	1	0	1

Como os registradores de entrada e saída (memória de dados) do microcontrolador ATmega são de 8 bits, os números com resolução de 8 bits são muito utilizados. Por exemplo, quando se quer escrever em um ou mais pinos se emprega um registrador PORTx, onde cada bit corresponde a um pino de I/O do microcontrolador (PXn). Considerando-se o registrador PORTD, e que todos os pinos foram configurados para serem saídas, se PORTD = 0b10101000 ou PORTD = 0xA8, isso significa que os pinos PD3, PD5 e PD7 vão ser colocados em nível lógico alto (VCC) e os demais em nível lógico zero (0 V):

	MSB				LSB			
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTD	PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
0b	1	0	1	0	1	0	0	0
0x	A				8			

Quando se quer escrever em um ou mais bits conhecidos, a notação binária é a mais adequada. Quando o número é conhecido, a notação hexadecimal é mais rápida de ser digitada por ser mais compacta.

MASCÁRA DE BITS E MACROS

Como os pinos do ATmega são tratados como bits individuais, o emprego de operações lógicas bit a bit são usuais e a linguagem C oferece suporte total a essas operações (tab. 4.4):

- | OU lógico bit a bit (usado para ativar bits , colocar em 1)
- & E lógico bit a bit (usado para limpar bits, colocar em 0)
- ^ OU EXCLUSIVO bit a bit (usado para trocar o estado dos bits)
- ~ complemento de 1 (1 vira 0, 0 vira 1)

Nr \gg x O número é deslocado x bits para a direita

Nr \ll x O número é deslocado x bits para a esquerda

Como o AVR-GCC não tem acesso direto a bits como o *assembly* (instruções SBI e CBI, detalhes no apêndice A), é necessário utilizar máscara de bits em operações com as variáveis. Isso é empregado quando se deseja alterar um ou mais bits de um registrador do ATmega ou de uma variável do C sem mudar os demais bits.

Exemplos para a **Variável_8bits = 0b10101000:**

1. Ativar o bit 0:

Variavel_8bits = Variavel_8Bits | 0b00000001;

$$\begin{array}{c} \text{0b10101000} \\ | \underline{\text{0b00000001}} \text{ (máscara)} \\ \text{Variavel_8bits} = \text{0b10101001} \end{array}$$

2. Limpar os bits 3 e 5:

Variavel_8bits = Variavel_8Bits & 0b11010111;

$$\begin{array}{c} \text{0b10101000} \\ \& \underline{\text{0b11010111}} \text{ (máscara)} \\ \text{Variavel_8bits} = \text{0b10000000} \end{array}$$

Para o trabalho exclusivo com bits individuais, declaram-se macros para que o compilador as utilize como se fossem funções. Essas macros trabalham com máscara e rotação de bits. Assim, o programador necessita criá-las e colocá-las nos seus programas.

As macros usuais são:

- **Ativação de bit, colocar em 1:**

```
#define set_bit(Y,bit_x) (Y|=(1<<bit_x))
```

onde Y |= (1<<bit_x) ou Y = Y | (1<<bit_x).

O programa ao encontrar a macro **set_bit(Y, bit_x)** irá substituí-la por **Y|=(1<<bit_x)**. Por exemplo, supondo que se deseje colocar o bit 5 do PORTD em 1 (pino PD5), a operação realizada é dada por **set_bit(PORTD,5)**. Dessa forma, o **Y** se transforma no PORTD e o **bit_x** é o número 5, resultando em:

```
PORTD = PORTD | (1<<5) ,
```

0xxxxxxxxx	(PORTD, x pode ser 0 ou 1)
0b00100000	(1<<5 é a máscara)
PORTD = 0bxx1xxxxx	(o bit 5 com certeza será 1)

A macro lê o valor PORTD e faz um OU bit a bit com o valor (1<<5) , 1 binário deslocado 5 bits para a esquerda, que é igual a 0b00100000, ou seja, o bit 5 é colocado em 1. Dessa forma independente do valor do PORTD, somente o seu bit 5 será colocado em 1 (OU lógico de x com 1 é igual a 1).

- **Limpeza de bit, colocar em 0:**

```
#define clr_bit(Y,bit_x) (Y&=~(1<<bit_x))
```

onde Y &= ~(1<<bit_x) ou Y = Y & (~ (1<<bit_x)).

O programa ao encontrar a macro **clr_bit(Y, bit_x)** irá substituí-la por **Y&=~(1<<bit_x)**. Por exemplo, supondo que se deseje zerar o bit 2 do PORTB (pino PB2), a operação realizada é dada por **clr_bit(PORTB,2)**. Assim, o **Y** se transforma no PORTB e o **bit_x** é o número 2, resultando em:

$\text{PORTB} = \text{PORTB} \& (\sim(1<<2)) ,$

0bxxxxxxxxx (PORTB, x pode ser 0 ou 1)
& 0b11111011 ($\sim(1<<2)$ é a máscara)
 $\text{PORTB} = 0bxxxxx0xx$ (o bit 2 com certeza será 0)

A macro lê o valor PORTB e faz um E bit a bit com o valor $\sim(1<<2)$, 1 binário deslocado 2 bits para a esquerda com o valor invertido dos seus bits, o que é igual a 0b11111011, ou seja, o bit 2 é colocado em 0. Dessa forma, independente do valor do PORTB, somente o seu bit 2 será colocado em 0 (E lógico de x com 0 é igual a 0).

- **Troca o estado lógico de um bit, 0 para 1 ou 1 para 0:**

`#define cpl_bit(Y,bit_x) (Y^=(1<<bit_x))`

onde $Y \wedge = (1<<bit_x)$ ou $Y = Y \wedge (1<<bit_x)$.

O programa ao encontrar a macro **cpl_bit(Y, bit_x)** irá substituí-la por **$Y^=(1<<bit_x)$** . Por exemplo, supondo que se deseje alterar o estado do bit 3 do PORTC (pino PC3), a operação realizada é dada por **cpl_bit(PORTC,3)**. Logo, o **Y** se transforma no PORTC e o **bit_x** é o número 3, resultando em:

$\text{PORTC} = \text{PORTC} \wedge (1<<3) ,$

0bxxxx1xxx (PORTC, x pode ser 0 ou 1)
^ 0b00001000 ($1<<3$ é a máscara)
 $\text{PORTC} = 0bxxxx0xxx$ (o bit 3 será 0 se o bit a ser complementado for 1 e 1 se ele for 0).

A macro lê o valor PORTC e faz um OU EXCLUSIVO bit a bit com o valor $(1<<3)$, 1 binário deslocado 3 bits para a esquerda, que é igual a 0b00001000, ou seja o bit 3 é colocado em 1. Dessa forma, independente do valor do PORTC, somente o seu bit 3 terá o seu estado lógico trocado (OU EXCLUSIVO de 0 com 1 é igual a 1, de 1 com 1 é igual a 0).

- **Leitura de um bit:**

```
#define tst_bit(Y,bit_x) (Y&(1<<bit_x))
```

O programa ao encontrar a macro **tst_bit(Y, bit_x)** irá substituí-la por **Y & (1<<bit_x)**. Por exemplo, supondo que se deseje ler o estado do bit 4 do PORTD (pino PD4), a operação realizada é dada por **tst_bit(PIND,4)**. Portanto, o **Y** se transforma no PIND e o **bit_x** é o número 4 (a leitura dos pinos do PORTD é feita no registrador PIND, ver o capítulo 5), resultando em:

```
PIND & (1<<4) ,  
  
          0bxxxxTxxxxx      (PIND, x pode ser 0 ou 1)  
          & 0b00010000      (1<<4 é a máscara)  
resultado = 0b000T0000      (o bit 4 terá o valor T, que será 0 ou 1).
```

A macro efetua apenas um operação E bit a bit de PIND com valor $(1 \ll 4)$, 1 binário deslocado 4 bits para a esquerda, que é igual a 0b00010000, ou seja o bit 4 é colocado em 1. Assim, somente o bit 3 do PIND mantém o seu valor. (E lógico de T com 1 é igual a T).

Na macro **tst_bit** não existe uma variável que recebe o valor da operação efetuada. Isso acontece porque a macro é utilizada com comandos de decisão tais como o **if** e o **while**. Por exemplo:

```
if(tst_bit(PIND,4))  
{  
    Ação a ser executada se o pino PD4 estiver em 1 lógico.  
}
```

A macro retorna 0 para o comando de decisão se o bit testado for 0 e um número diferente se ele estiver em 1. Como os comandos de decisão em C executam a condição entre as chaves $\{\}$ se a sua condição de teste for verdadeira, ela será verdade para valores diferentes de zero.

4.5.10 PRODUZINDO UM CÓDIGO EFICIENTE

Quando se trabalha com microcontroladores, existe uma dependência entre o desempenho e a forma como o código é criado. As facilidades da linguagem de programação C, usuais na programação de um computador, devem ser empregadas com cuidado, pois se programará um microcontrolador com uma memória bem menor, tanto de programa, como RAM. Além do que, um microcontrolador pode possuir um núcleo de poucos bits, como é o caso do ATmega, de 8 bits.

Dessa forma, é importante gerar um código pensando nas limitações do microcontrolador e nas características do compilador empregado. A Atmel disponibiliza um manual de aplicação¹⁸ especialmente com essas questões, o qual é resumido a seguir.

Reduzindo o tamanho do código compilado

1. Compile com a máxima otimização possível.
2. Use variáveis locais sempre que possível.
3. Use o menor tipo de dado possível (8 bits), **unsigned** se aplicável.
4. Se uma variável não local é referenciada somente dentro de uma função, ela deve ser declarada como **static**.
5. Junte variáveis não-locais em estruturas, se conveniente. Isso aumenta a possibilidade de endereçamento indireto sem recarga de ponteiro.
6. Para acessar memória mapeada, use ponteiros.
7. Use **do{ } while(expressão)** se aplicável.
8. Use laços com contadores decrescentes e pré-decrementados, se possível.
9. Acesse a memória de I/O diretamente (não use ponteiros).
10. Use *macros* ao invés de funções para tarefas menores que 2-3 linhas de código em *assembly*.

¹⁸ Application note: AVR035: Efficient C Coding for AVR. Recomenda-se fortemente a sua leitura.

11. Evite chamar funções dentro de interrupções.
12. Se possível junte várias funções em um módulo (biblioteca), para aumentar o reuso do código.
13. Em alguns casos o emprego da compilação com otimização de velocidade pode resultar em códigos menores que a compilação para gerar o menor código (é bom testar ambas).

Reduzindo o tamanho da memória RAM necessária

1. Todas as constantes e literais devem ser colocados na memória *flash* (de programa).
2. Evite usar variáveis globais. Empregue variáveis locais sempre que possível (isso também economiza a memória de programa).

Uma técnica interessante, quando se aprende a programar, é alterar o código sem mudar a sua funcionalidade e compilá-lo novamente, verificando qual alteração resultou em um menor número de bytes.

Todo bom programa deve ser adequadamente comentado e organizado.

Códigos eficientes são resultantes de bons algoritmos, produzindo maior densidade de código (funcionalidade/bytes). É fundamental conhecer a arquitetura interna do microcontrolador para desenvolver melhores programas.

4.5.11 ESTRUTURAÇÃO E ORGANIZAÇÃO DO PROGRAMA

A estruturação do programa é um requisito para a sua portabilidade e manutenção. A princípio todo código pode ser escrito em um único arquivo fonte. Entretanto, para projetos maiores é aconselhável sua divisão em vários arquivos organizados de acordo com suas funcionalidades. Isso é feito criando-se um arquivo *.h (*header file*) para cada arquivo *.c individual. O padrão é colocar nos arquivos as seguintes informações:

Arquivo *.h

Includes, defines, typedef, protótipo de funções.

Arquivo *.c

Código das funções e declarações das variáveis utilizadas. No início do arquivo é feito a inclusão do(s) arquivo(s) com extensão h utilizado(s).

Na fig. 4.6, é apresentado um exemplo de um projeto chamado **fonte_digital**. O programa principal utiliza dois arquivos com funções, o LCD.c e ADC.c e os seus respectivos *.h.

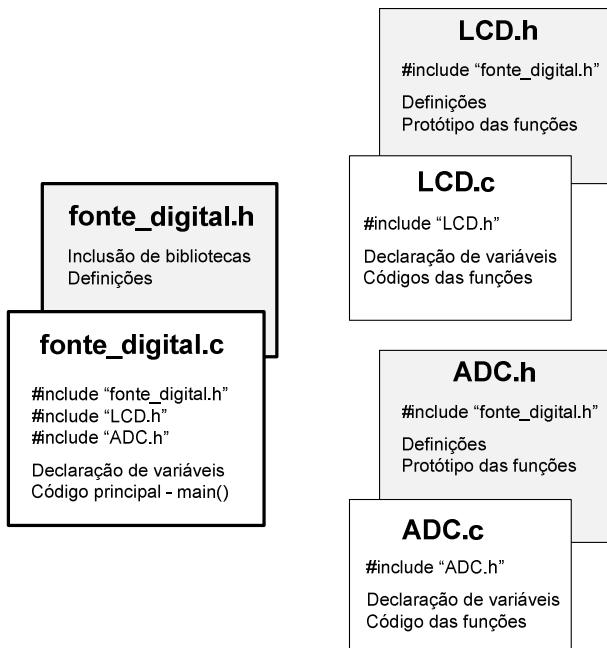


Fig. 4.6 – Exemplo da estruturação de um projeto com arquivos de programa separados.

A IDE de programação permite a inclusão desses arquivos no projeto principal. Exemplos com o AVR Studio serão vistos nos capítulos posteriores.

O compilador faz a compilação individual dos arquivos *.c gerando arquivos com extensão .o (objeto), depois a montagem os interconecta de acordo com a escrita do programa e gera um único arquivo *.hex (ou *.exe no caso de um programa para computador).

A grande vantagem da divisão do programa em arquivos individuais é a facilidade de correções e mudanças nas definições. Além, de permitir o desenvolvimento de bibliotecas de funções próprias e a portabilidade do código. A estruturação é a filosofia empregada para a programação orientada a objeto e utilizada por programadores e equipes de programação profissionais.

5. PORTAS DE ENTRADA E SAÍDA (I/Os)

A primeira dúvida ao se trabalhar com um microcontrolador é como funcionam os seus pinos, ou seja, com são escritos e lidos dados nas suas entradas e saídas. Neste capítulo, é feita uma introdução à programação do ATmega328, começando com os conceitos de mais fácil compreensão: a leitura e escrita de seus pinos de I/O. O estudo inicia com o acionamento de LEDs e a leitura de botões, seguido das técnicas para o uso de displays de 7 segmentos e de cristal líquido (LCD 16×2). Também se explica o emprego de rotinas de atraso, comuns na programação microcontrolada.

O ATmega328 possui 3 conjuntos de pinos de entrada e saída (I/Os): PORTB, PORTC e PORTD; respectivamente, pinos PB7 .. PB0, PC6 .. PC0 e PD7 .. PD0, todos com a função Lê – Modifica – Escreve. Isso significa que a direção de um pino pode ser alterada sem mudar a direção de qualquer outro pino do mesmo PORT (instruções SBI e CBI¹⁹). Da mesma forma, os valores lógicos dos pinos podem ser alterados individualmente, bem como a habilitação dos resistores de *pull-up* para os pinos configurados como entrada. Cada PORT possui um registrador de saída com características simétricas, isto é, com a mesma capacidade para drenar ou suprir corrente, suficiente para alimentar LEDs diretamente (20 mA por pino). O cuidado a se ter é respeitar a máxima corrente total que o componente e que cada PORT suporta, 200 mA e 100 mA, respectivamente²⁰. Todas os pinos têm resistores de *pull-up* internamente e diodos de proteção entre o VCC e o terra, além de uma capacitância de 10 pF, como indicado na fig. 5.1.

¹⁹ SBI = ativa um bit (coloca em 1). CBI = limpa um bit (coloca em 0).

²⁰ Sempre que possível deve-se utilizar a menor corrente. É importante consultar o manual do fabricante para a observação das características elétricas do microcontrolador.

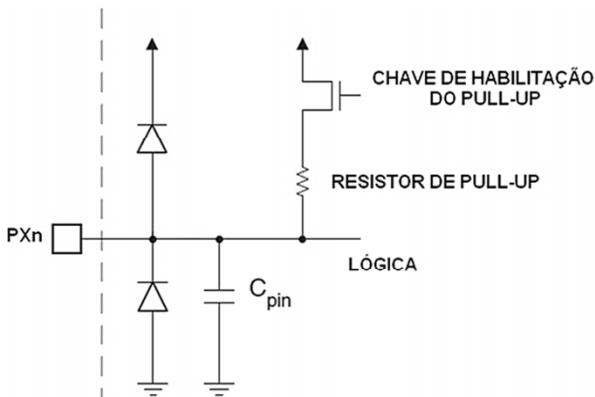


Fig. 5.1 – Esquema geral dos pinos de I/O (PXn) do ATmega.

Os registradores responsáveis pelos pinos de I/O são:

- **PORTx:** registrador de dados, usado para escrever nos pinos do PORTx.
- **DDRx:** registrador de direção, usado para definir se os pinos do PORTx são entrada ou saída.
- **PINx:** registrador de entrada, usado para ler o conteúdo dos pinos do PORTx.

Em resumo, para o uso de um pino de I/O, deve-se primeiro definir se ele será entrada ou saída escrevendo-se no registrador DDRx. Então, a escrita no registrador PORTx alterará o estado lógico do pino se ele for saída, ou poderá habilitar o *pull-up* interno, se ele for entrada²¹. Os estados lógicos dos pinos do PORT são lidos do registrador PINx. É importante notar que para a leitura do PINx logo após uma escrita no PORTx ou DDRx, deve ser gasto pelo menos um ciclo de máquina para a sincronização dos dados pelo microcontrolador.

²¹ Se algum pino não for utilizado é recomendado que o seu nível lógico seja definido. Entradas com nível flutuante devem ser evitadas para evitar o consumo de corrente quando o pino não estiver sendo empregado. Neste caso a Atmel recomenda a habilitação do *pull-up* (para maiores detalhes ver o manual do fabricante).

Na tab. 5.1, é apresentada as configurações dos bits de controle dos registradores responsáveis pela definição do comportamento dos pinos (ver a tab. 2.1 – registradores de entrada e saída do ATmega328).

Tab. 5.1 - Bits de controle dos pinos dos PORTs.

DDXn [*]	PORTXn	PUD (no MCUCR)	I/O	Pull-up	Comentário
0	0	x	Entrada	Não	Alta impedância (Hi-Z).
0	1	0	Entrada	Sim	PXn irá fornecer corrente se externamente for colocado em nível lógico 0.
0	1	1	Entrada	Não	Alta impedância (Hi-Z).
1	0	x	Saída	Não	Saída em zero (drena corrente).
1	1	x	Saída	Não	Saída em nível alto (fornecce corrente).

*X = B, C ou D; n = 0, 1, ... ou 7.

Quando o bit PUD (Pull-Up Disable) no registrador MCUCR (MCU Control Register) está em 1 lógico, os pull-ups em todos os PORTs são desabilitados, mesmo que os bits DDXn e PORTXn estejam configurados para habilitá-los.

5.1 ROTINAS SIMPLES DE ATRASO

Rotinas de atraso são muito comuns na programação de microcontroladores. São realizadas fazendo-se a CPU gastar ciclos de máquina na repetição de instruções. Para se calcular o exato número de ciclos de máquina gastos em uma sub-rotina de atraso é necessário saber quantos ciclos cada instrução consome. Para exemplificar é utilizado o programa *assembly* a seguir.

Atraso:

```

DEC R3      //decrementa R3, começa com o valor 0x00
BRNE Atraso //enquanto R3 > 0 fica decrementando R3, desvio para Atraso
DEC R2      //decrementa R2, começa com o valor 0x00
BRNE Atraso //enquanto R2 > 0 volta a decrementar R3
RET         //retorno da sub-rotina

```

No programa acima, a instrução DEC consome 1 ciclo, a instrução BRNE consome 2 ciclos e na última vez, quando não desvia mais, consome 1 ciclo. Como os registradores R3 e R2 possuem o valor zero inicialmente²² (o primeiro decremento os leva ao valor 255) e o decreimento de R3 é repetido dentro do laço de R2, espera-se que haja 256 decrementos de R3 vezes 256 decrementos de R2. Se for considerado 3 ciclos para DEC e BRNE, tem-se aproximadamente $(3 \times 256 \times 256) + (3 \times 256)$ ciclos, ou seja, 197.376 ciclos. A parcela 3×256 , que é o tempo gasto no decremento de R2, pode ser desprezada, pois afeta pouco o resultado final. Assim, a fórmula aproximada seria dada por $3 \times 256 \times 256$, igual a 196.608 ciclos.

Entretanto, BRNE não consome dois ciclos no último decremento e sim um. Desta forma, o cálculo preciso é mais complexo:

$$((3 \text{ ciclos} \times 255) + 2 \text{ ciclos}) + 3 \text{ ciclos} \times 255 + 769 \text{ ciclos} = 197.119 \text{ ciclos.}$$

Para melhor compreensão, observar a fig. 5.2. Se forem considerados os ciclos gastos para a chamada da sub-rotina, com a instrução RCALL (3 ciclos) e os ciclos para o retorno, com a instrução RET (4 ciclos), tem-se um gasto total da chamada da sub-rotina até seu retorno de 197.126 ciclos.

Atraso:

$$\begin{aligned} \text{DEC R3} & \left[\begin{smallmatrix} +1 \text{ ciclo} \\ 2 \text{ ciclos} \end{smallmatrix} \right] \times 255 + \left[\begin{smallmatrix} 1 \text{ ciclo} \\ 1 \text{ ciclo} \end{smallmatrix} \right] = \left[\begin{smallmatrix} 767 \text{ ciclos} \\ +1 \text{ ciclo} \\ +2 \text{ ciclos} \end{smallmatrix} \right] \\ \text{BRNE Atraso} & \\ \text{DEC R2} & \\ \text{BRNE Atraso} & \left[\begin{smallmatrix} 767 \text{ ciclos} \\ +1 \text{ ciclo} \\ +1 \text{ ciclo} \end{smallmatrix} \right] = 197119 \text{ ciclos} \end{aligned}$$

Fig. 5.2 – Cálculo preciso da sub-rotina de atraso.

O tempo gasto pelo microcontrolador dependerá da frequência de trabalho utilizada. Como no AVR um ciclo de máquina equivale ao inverso da frequência do *clock* (período), o tempo gasto será dado por:

²² Após a inicialização do ATmega os valores dos registradores R0:R31 são indeterminados, considera-se zero para a simplificação do programa. Caso seja necessário que os registradores tenham um valor conhecido, eles devem ser inicializados. Uma vez executada a sub-rotina de atraso e considerando-se que os registradores empregados nela não são usados em outras partes do programa, eles sempre estarão inicialmente em zero. Na dúvida, os registradores sempre devem ser inicializados.

$$Tempo\ Gasto = N^{\circ}\ de\ ciclos \times \frac{1}{Freq.\ de\ trabalho} \quad (5.1)$$

Logo, para o exemplo acima, com um *clock* de 16 MHz (período de 62,5 ns), da chamada da sub-rotina até seu retorno, resulta em:

$$Tempo\ Gasto = 197.126 \times 62,5\ ns = 12,32\ ms$$

Exercícios:

5.1 – Qual o tempo aproximado e exato gasto pelo ATmega para a execução da sub-rotina abaixo?

ATRASO:

```
LDI R19,X      //carrega R19 com o valor X ( dado abaixo)
volta:
DEC R18        //decrementa R18, começa com 0x00
BRNE volta     //enquanto R18 > 0 volta a decrementar R18
DEC R19        //decrementa R19
BRNE volta     //enquanto R19 > 0 vai para volta
RET
```

a) X = 0, f_{clk} = 16MHz.

b) X = 10, f_{clk} = 1MHz.

5.2 – Qual o tempo aproximado gasto para a execução da sub-rotina abaixo para uma frequência de operação do ATmega de 8 MHz (fluxograma na fig. 4.4b)?

ATRASO:

```
LDI R19,0x02
volta:
DEC R17        //decrementa R17, começa com 0x00
BRNE volta     //enquanto R17 > 0 fica decrementando R17
DEC R18        //decrementa R18, começa com 0x00
BRNE volta     //enquanto R18 > 0 volta a decrementar R18
DEC R19        //decrementa R19
BRNE volta     //enquanto R19 > 0 vai para volta
RET
```

5.3 – Desenvolva uma sub-rotina em *assembly* para produzir um atraso de aproximadamente 0,5 s para o ATmega operando a 16 MHz.

5.4 – Para o programa abaixo, escrito na linguagem C, qual é o tempo aproximado gasto pelo ATmega para a sua execução ($f_{clk} = 20$ MHz). Considere que são gastos 3 ciclos de *clock* para que uma repetição do laço **for** seja realizada.

```
unsigned int i, j;  
  
for(i=256; i!=0; i--)  
{  
    for(j=65535; j!=0; j--);  
}
```

5.2 LIGANDO UM LED

Para começar o trabalho com o ATmega, será feito o acionamento de um LED. Como o Arduino possui um LED ligado diretamente ao pino PB5 do ATmega, não é necessário o uso de um circuito adicional (fig. 5.3).

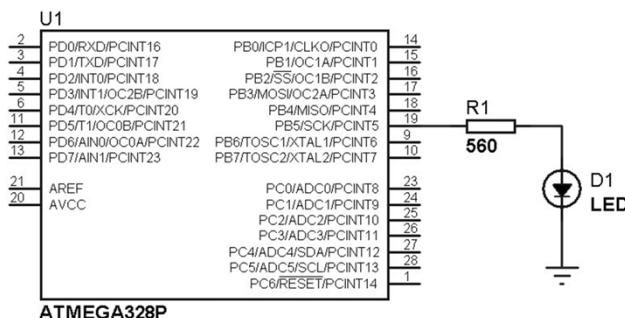


Fig. 5.3 – Ligando um LED.

Como o microcontrolador trabalha de acordo com um programa, por mais simples que pareça ligar um LED, existe uma infinidade de possibilidades: o LED pode ser piscado, a frequência pode ser alterada, o número de vezes que se liga e desliga pode ser ajustada, bem como o tempo de acionamento.

Quando se aprende a programar um microcontrolador, o primeiro programa é piscar um LED, tal como o “Hello Word!” quando se programa um computador. Para isso basta ligar o LED, esperar um tempo, desligar o

LED, esperar um tempo e voltar novamente a ligá-lo, repetindo o processo indefinidamente. O programa em *assembly* para o Arduino que executa essa tarefa é dado a seguir (o fluxograma do programa pode ser visto na fig. 4.4a).

Pisca_LED.asm

```
-----  
.equ LED    = PB5      //LED é o substituto de PB5 na programação  
.ORG 0x000      //endereço de início de escrita do código  
  
INICIO:  
    LDI R16,0xFF      //carrega R16 com o valor 0xFF  
    OUT DDRB,R16      //configura todos os pinos do PORTB como saída  
  
PRINCIPAL:  
    SBI PORTB, LED    //coloca o pino PB5 em 5V  
    RCALL ATRASO      //chama a sub-rotina de atraso  
    CBI PORTB, LED    //coloca o pino PB5 em 0V  
    RCALL ATRASO      //chama a sub-rotina de atraso  
    RJMP PRINCIPAL    //volta para PRINCIPAL  
  
ATRASO:                 //atraso de aprox. 200 ms  
    LDI R19,16  
volta:  
    DEC R17            //decrementa R17, começa com 0x00  
    BRNE volta          //enquanto R17 > 0 fica decrementando R17  
    DEC R18            //decrementa R18, começa com 0x00  
    BRNE volta          //enquanto R18 > 0 volta a decrementar R18  
    DEC R19            //decrementa R19  
    BRNE volta          //enquanto R19 > 0 vai para volta  
    RET  
-----
```

O programa *assembly* consumiu 30 bytes de memória de programa (*flash*) com o uso de 15 instruções, todas de 16 bits.

O programa escrito na linguagem C é apresentado a seguir. É importante no AVR *Studio* configurar a compilação para a otimização máxima, atalho *<Alt + F7>*, ver a fig. 3.12. Após a montagem²³, o programa resultou em 178 bytes de memória de programa (as vantagens e desvantagens do *assembly* sobre o C e vice-versa foram apresentadas no capítulo 4).

²³ Ver nota de rodapé de número 12 na página 46.

Pisca_LED.c

```
-----  
#define F_CPU 16000000UL /*define a frequência do microcontrolador 16MHz  
                         (necessário para usar as rotinas de atraso)*/  
#include <avr/io.h>      //definições do componente especificado  
#include <util/delay.h>   /*biblioteca para o uso das rotinas de  
                         _delay_ms() e _delay_us()*/  
  
//Definições de macros - empregadas para o trabalho com os bits  
#define set_bit(Y,bit_x) (Y|=(1<<bit_x))    /*ativa o bit x da  
                                              variável Y (coloca em 1)*/  
#define clr_bit(Y,bit_x) (Y&=~(1<<bit_x))    /*limpa o bit x da variável Y  
                                              (coloca em 0)*/  
#define tst_bit(Y,bit_x) (Y&(1<<bit_x))     /*testa o bit x da variável Y  
                                              (retorna 0 ou 1)*/  
#define cpl_bit(Y,bit_x) (Y^=(1<<bit_x))     /*troca o estado do bit x da  
                                              variável Y (complementa)*/  
  
#define LED PB5 //LED é o substituto de PB5 na programação  
  
int main( )  
{  
    DDRB = 0xFF; //configura todos os pinos do PORTB como saídas  
  
    while(1)          //laço infinito  
    {  
        set_bit(PORTB,LED); //liga LED  
        _delay_ms(200);    //atraso de 200 ms  
        clr_bit(PORTB,LED); //desliga LED  
        _delay_ms(200);    //atraso de 200 ms  
    }  
}  
-----
```

Para compreender o programa acima, é importante dominar o trabalho com bits (ver a seção 4.5.9). Esse conhecimento é fundamental para se programar eficientemente o ATmega.

O acionamento de um LED é uma das tarefa mais simples que se pode realizar com um microcontrolador. LEDs sinalizadores estão presentes em quase todos os projeto eletrônicos e pode-se dar muito maior complexidade a função de sinalização, como por exemplo, indicar a carga de processamento da CPU, aumentando-se ou diminuindo-se a frequência com que um LED é ligado e desligado.

Exercícios:

5.5 – Faça um programa em *assembly* para piscar um LED a cada 1 s.

5.6 – Utilizando a macro para complementar um bit (`cpl_bit`), faça um programa para piscar um LED a cada 500 ms.

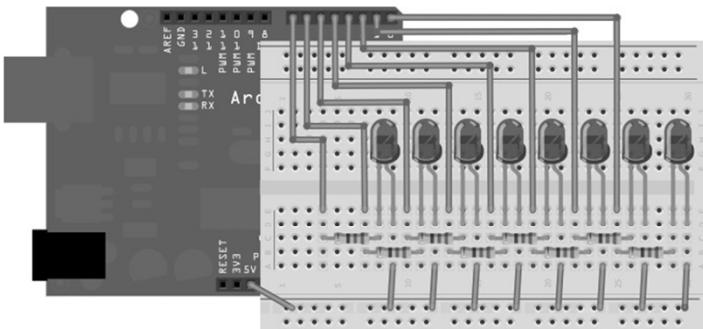
5.7 – Desenvolva um programa para piscar um LED rapidamente 3 vezes e 3 vezes lentamente.

5.8 – Utilizando o deslocamento de bits crie um programa em *assembly* que ligue 8 LEDs²⁴ (ver a fig. 5.4a), da seguinte forma:

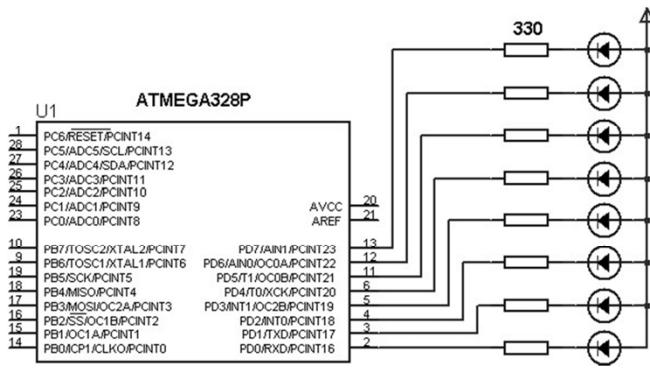
- a) Ligue sequencialmente 1 LED da direita para a esquerda (o LED deve permanecer ligado até que todos os 8 estejam ligados, depois eles devem ser desligados e o processo repetido).
- b) Ligue sequencialmente 1 LED da esquerda para a direita, mesma lógica da letra a.
- c) Ligue sequencialmente 1 LED da direita para a esquerda, desta vez somente um LED deve ser ligado por vez.
- d) Ligue sequencialmente 1 LED da esquerda para a direita e vice-versa (vai e volta), só um LED deve ser ligado por vez.
- e) Ligue todos os LEDs e apague somente 1 LED de cada vez, da direita para a esquerda e vice-versa (vai e volta), somente um LED deve ser apagado por vez.
- f) Mostre uma contagem binária crescente (0-255) com passo de 250 ms.
- g) Mostre uma contagem binária decrescente (255-0) com passo de 250 ms.

²⁴ Como o Arduino utiliza um programa de *boot loader*, ele emprega os pinos TXD e RXD para a gravação da memória de programa (pinos PD0 e PD1). Quando não se utiliza a IDE do Arduino para a gravação, esse pinos devem ser configurados explicitamente pelo programador para serem pinos de I/O genéricos, caso contrário os pinos não se comportarão como esperado (configuração *default*). Em C é necessário acrescentar a seguinte linha de código: `UCSR0B = 0x00;` //desabilita RXD e TXD.

Importante: quando se grava o Arduino, os pinos 0 e 1 (PD0 e PD1) não devem estar conectados a nenhum circuito, caso contrário, poderá haver erro de gravação. Se algum *shield* utilizar os referidos pinos, é aconselhável gravar primeiro o Arduino e somente depois conectá-lo.



a)



b)

Fig. 5.4 – Sequencial com 8 LEDs: a) montagem para o Arduino e b) esquemático.

5.3 LENDO UM BOTÃO (CHAVE TÁCTIL)

Quando se começa o estudo de um microcontrolador, além de se ligar um LED, um dos primeiros programas é liga-lo ao se pressionar um botão (tecla ou chave táctil). O problema é que na prática, botões apresentam o chamado *bounce*, um ruído que pode ocorrer ao se pressionar ou soltar o botão. Esse ruído produz uma oscilação na tensão proveniente do botão, ocasionando sinais lógicos aleatórios que podem produzir leituras errôneas. Se o ruído existir, geralmente ele desaparece após

aproximadamente 10 ms. Esse tempo depende das características físicas e elétricas do botão e do circuito onde ele se encontra.

Dependendo da configuração do circuito do botão, o seu pressionar pode resultar em um sinal lógico alto ou baixo. Isso depende da forma como se emprega o resistor necessário para a leitura do botão: com *pull-up*, entre a alimentação (VCC) e o botão, ou com *pull-down*, entre o botão e o terra. Na fig. 5.5, é exemplificado o ruído que pode ser produzido por um botão com essas configurações.

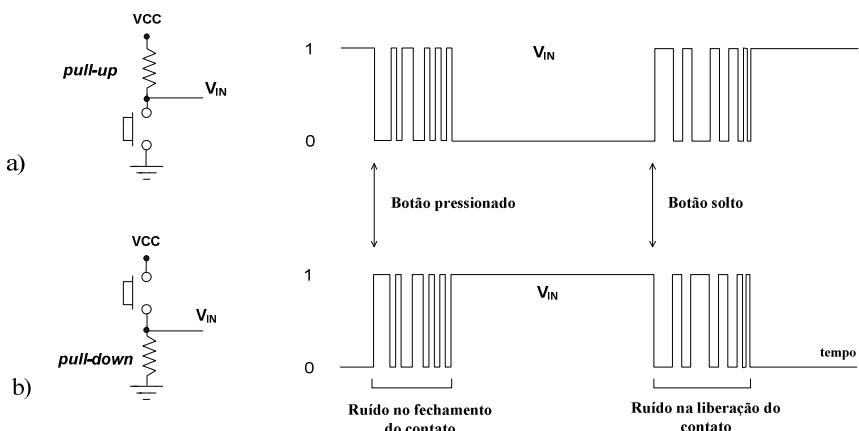


Fig. 5.5 – Exemplo do ruído que pode ser gerado ao se pressionar e soltar um botão:
a) usando um resistor de *pull-up* e b) usando um resistor de *pull-down*.

O ruído no fechamento do contato, geralmente não produz problemas, pois muitas vezes não existe ou é muito pequeno para ser notado. O ruído na liberação do contato é comum e possui maior duração devido, principalmente, ao contato mecânico do botão. Quando se pressiona o botão, os seus contatos são mantidos pela pressão aplicada sobre ele; quando ele é solto, o contato se desfaz e existe um repique mecânico. Além disso, o circuito do botão possui intrinsecamente indutâncias e capacitâncias que geram ruídos elétricos quando o circuito é fechado ou aberto.

Na fig. 5.6, é apresentado o ruído obtido experimentalmente quando um botão (comum em eletrônica) com um *pull-up* de $10\text{ k}\Omega$ e ligado a 5 V é liberado (avaliado com o emprego de um osciloscópio). Pode-se notar que o ruído tem duração de aproximadamente 3 ms (divisão horizontal de $50\text{ }\mu\text{s}$ e vertical de 1 V).

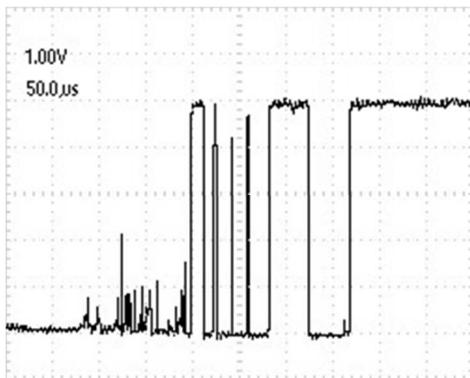


Fig. 5.6 – Ruído real gerado ao se soltar um botão com *pull-up*.

Observando-se o ruído da fig. 5.6 e sabendo-se que a velocidade de operação de um sistema digital depende da sua frequência de trabalho, quanto maior a frequência de leitura do botão mais sensível ao ruído o sistema se torna. Para resolver esse problema é necessário o uso de capacitores ou outros componentes eletrônicos. Todavia, ele é facilmente resolvido quando o sistema é controlado por um programa.

A técnica de hardware para eliminação do *bounce* é o chamado *debounce*. Em sistemas microcontrolados, o *debounce* é feito via software, sem a necessidade de componentes externos para a filtragem do ruído.

A seguir, é apresentado o fluxograma para um programa que troca o estado de um LED toda vez que um botão é pressionado (fig. 5.7). O mesmo emprega um pequeno tempo para eliminar o eventual ruído do botão quando solto. O ruído no pressionamento não é considerado. O circuito empregado é apresentado na fig. 5.8. Como o ATmega possui resistores

internos de *pull-up*, para a leitura de um botão basta somente ligá-lo diretamente ao terra e a um dos pinos de I/O do microcontrolador.

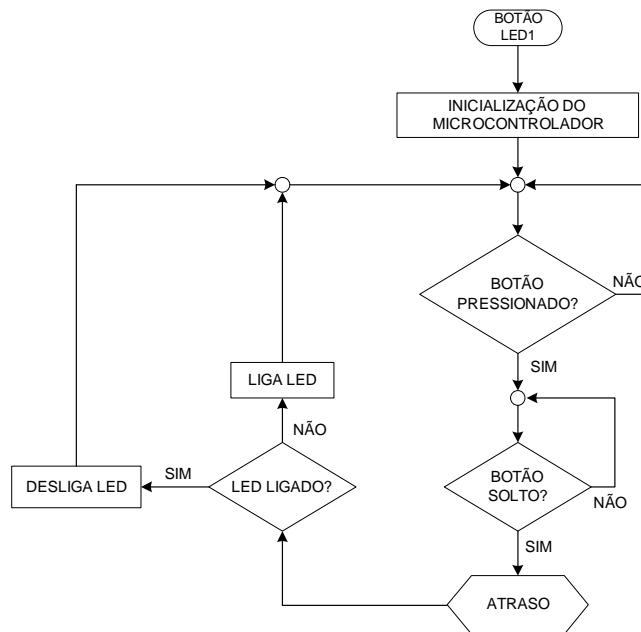
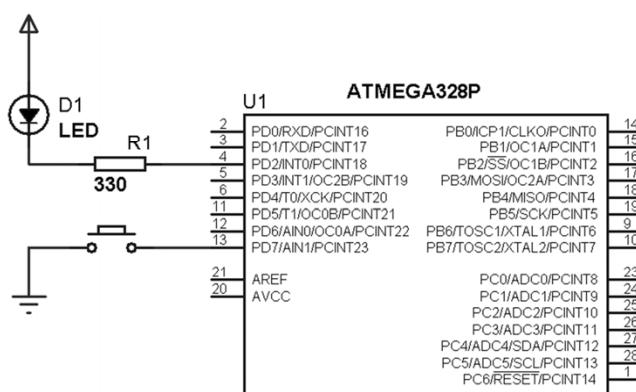
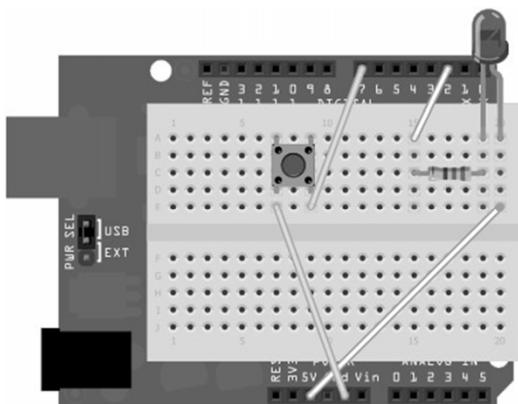


Fig. 5.7 – Fluxograma do programa para ligar e apagar um LED com um botão.



a)



b)

Fig. 5.8 – Circuito para ligar e apagar um LED com um botão: a) esquemático e b) montagem para o Arduino.

O programa em *assembly* é apresentado a seguir (ver o apêndice A para detalhes das instruções *assembly* do ATmega328). É bom compará-lo com o fluxograma da fig. 5.7 para sua compreensão. Também é importante observar que algumas instruções em *assembly* só trabalham com alguns dos 32 registradores de propósito geral. Após a montagem, o tamanho do código foi de 42 bytes (21 instruções).

Botao_LED.asm

```
===== //  
// LIGANDO E DESLIGANDO UM LED QUANDO UM BOTÃO É PRESSIONADO //  
===== //  
//DEFINIÇÕES  
.equ LED = PD2 //LED é o substituto de PD2 na programação  
.equ BOTAO = PD7 //BOTAO é o substituto de PD7 na programação  
.def AUX = R16 /*R16 tem agora o nome de AUX (nem todos os 32 registradores  
de uso geral podem ser empregados em todas as instruções) */  
-----  
.ORG 0x000 /*endereço de início de escrita do código na memória flash,  
após o reset o contador do programa aponta para cá.*/  
Inicializacoes:  
    LDI AUX,0b00000100 //carrega AUX com o valor 0x04 (1 = saída e 0 = entrada)  
    OUT DDRD,AUX      //configura PORTD, PD2 saída e demais pinos entradas  
    LDI AUX,0b11111111 /*habilita o pull-up para o botão e apaga o LED (pull-up em  
todas as entradas)*/  
    OUT PORTD,AUX  
  
    NOP      /*sincronização dos dados do PORT. Necessário somente para  
uma leitura imediatamente após uma escrita no PORT*/
```

```

//-----
//LAÇO PRINCIPAL
//-----
Principal:
    SBIC  PIND,BOTA0      //verifica se o botão foi pressionado, senão
    RJMP  Principal        //volta e fica preso no laço Principal

Esp_Soltar:
    SBIS  PIND,BOTA0      //se o botão não foi solto, espera soltar
    RJMP  Esp_Soltar
    RCALL Atraso          /*após o botão ser solto gasta um tempo para eliminar o
                           ruido proveniente do mesmo*/
    SBIC  PORTD,LED       //se o LED estiver apagado, liga e vice-versa
    RJMP  Liga
    SBI   PORTD,LED       //apaga o LED
    RJMP  Principal        //volta ler botão

Liga:
    CBI   PORTD,LED       //liga LED
    RJMP  Principal        //volta ler botão

//-----
//SUB-ROTINA DE ATRASO - Aprox. 12 ms a 16 MHz
//-----
Atraso:
    DEC   R3              //decrementa R3, começa com 0x00
    BRNE  Atraso           //enquanto R3 > 0 fica decrementando R3
    DEC   R2              //enquanto R2 > 0 volta a decrementar R3
    BRNE  Atraso
    RET
//=====

```

Muitos microcontroladores da família AVR necessitam que o *Stack Pointer* seja inicializado pelo programa em *assembly* (em C a inicialização é feita automaticamente pelo compilador). Para o ATmega328 isso não é necessário, mas caso desejado, ela poderia ser feita com:

```

//inicialização do Stack Pointer deve ser feita com o endereço final da SRAM
LDI  R16, high(RAMEND)
OUT SPH, R16             //registrador SPH = parte alta do endereço
LDI  R16, low(RAMEND)
OUT SPL, R16              //registrador SPL = parte baixa do endereço

```

Para comparação com o programa *assembly*, é apresentado a seguir o código em C com a mesma funcionalidade (fluxograma da fig. 5.7).

Botao_LED.c

```
//===================================================================== //
//      LIGANDO E DESLIGANDO UM LED QUANDO UM BOTÃO É PRESSIONADO      //
//===================================================================== //
#define F_CPU 16000000UL /*define a frequência do microcontrolador 16MHz (necessário
                        para usar as rotinas de atraso)*/
#include <avr/io.h>      //definições do componente especificado
#include <util/delay.h>  //bibliot. para as rotinas de _delay_ms() e delay_us()

//Definições de macros - para o trabalho com os bits de uma variável
#define set_bit(Y,bit_x)(Y|=(1<<bit_x)) //ativa o bit x da variável Y (coloca em 1)
#define clr_bit(Y,bit_x)(Y&=~(1<<bit_x)) //limpa o bit x da variável Y (coloca em 0)
#define cpl_bit(Y,bit_x)(Y^=(1<<bit_x)) //troca o estado do bit x da variável Y
#define tst_bit(Y,bit_x)(Y&(1<<bit_x)) //testa o bit x da variável Y (retorna 0 ou 1)

#define LED PD2 //LED é o substituto de PD2 na programação
#define BOTA0 PD7 //BOTA0 é o substituto de PD7 na programação
//-----
int main()
{
    DDRD = 0b00000100; //configura o PORTD, PD2 saída, os demais pinos entradas
    PORTD= 0b11111111; /*habilita o pull-up para o botão e apaga o LED (todas as
                        entradas com pull-ups habilitados)*/
    while(1) //laço infinito
    {
        if(!tst_bit(PIND,BOTA0))//se o botão for pressionado executa o if
        {
            while(!tst_bit(PIND,BOTA0)); //fica preso até soltar o botão
            _delay_ms(10); //atraso de 10 ms para eliminar o ruído do botão

            if(tst_bit(PORTD,LED)) //se o LED estiver apagado, liga o LED
                clr_bit(PORTD,LED);
            else //se não apaga o LED
                set_bit(PORTD,LED);

            //o comando cpl_bit(PORTD,LED) pode substituir este laço if-else
        }//if do botão pressionado
    }//laço infinito
}
//=====================================================================
```

Outra variante da leitura de botões, é executar o que necessita ser feito imediatamente após o botão ser pressionado e avaliar se o botão já foi solto. Após isso, então, se gasta um pequeno tempo para eliminar o *bounce*. O fluxograma da fig. 5.9 apresenta essa ideia.

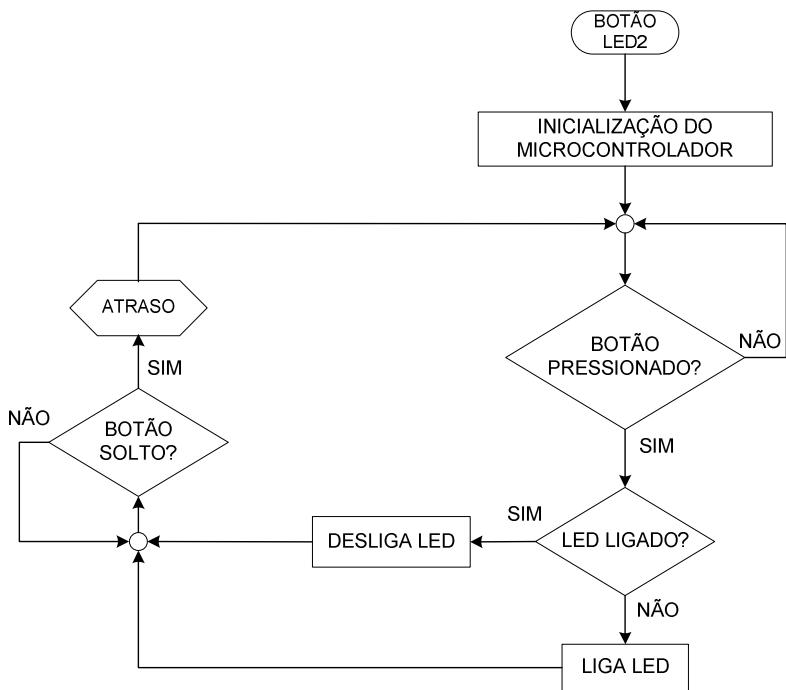


Fig. 5.9 – Fluxograma para ligar ou desligar imediatamente um LED após um botão ser pressionado.

Quando se necessita executar repetidamente uma determinada ação, poderia ser o incremento de uma variável, por exemplo, outra maneira de leitura de um botão deve ser empregada. Nesse caso, a leitura do botão ficará sempre dentro de um laço com uma rotina de tempo adequada para a realização da tarefa. O ruído é eliminado pelo tempo gasto. O fluxograma da fig. 5.10 apresenta essa ideia utilizada para piscar um LED enquanto um botão é mantido pressionado.

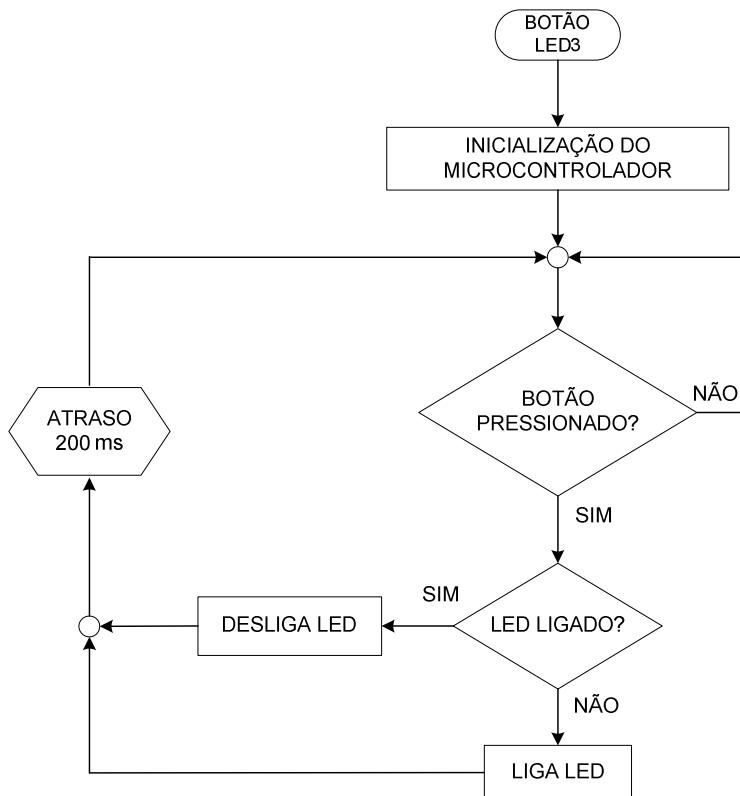


Fig. 5.10 – Fluxograma para piscar um LED enquanto um botão é mantido pressionado.

Em resumo, existem três formas para o *debounce* por software:

1. Após o botão ser pressionado, esperar o mesmo ser solto e gastar um pequeno tempo para que o ruído desapareça e, então, efetuar a ação correspondente.
2. Após o botão ser pressionado, efetuar imediatamente a ação correspondente, esperar o botão ser solto e depois esperar um pequeno tempo.
3. Após o botão ser pressionado, efetuar imediatamente a ação correspondente e depois esperar um tempo adequado para repetir a leitura do botão.

Se houver ruído quando o botão é pressionado, um atraso para eliminá-lo pode ser necessário. Todavia, a lógica para o *debounce* não muda. Dependendo da ação que será feita ao se pressionar o botão, gastar um determinado tempo para o *debounce* pode ser desnecessário. Isso dependerá somente do tempo que a ação leva para ser executada e do tempo para uma nova leitura do botão.

Exercícios:

- 5.9** – Elaborar um programa para ligar imediatamente um LED após o pressionar de um botão, com uma rotina de atraso de 10 ms para eliminação do *bounce*.
- 5.10** – Elaborar um programa que troque o estado do LED se o botão continuar sendo pressionado. Utilize uma frequência que torne agradável o piscar do LED.
- 5.11** – Elaborar um programa para aumentar a frequência em que um LED liga e desliga, enquanto um botão estiver sendo pressionado, até o momento em que o LED ficará continuamente ligado. Quando o botão é solto o LED deve ser desligado.
Qual a aplicação prática dessa técnica, imaginando que o botão pode ser o sinal proveniente de algum sensor e o LED algum dispositivo sinalizador?
- 5.12** – No exercício 5.8, foram propostas 7 animações com 8 LEDs. Crie outra animação, totalizando 8. Depois empregue dois botões: um será o AJUSTE, que quando pressionado permitirá que o outro botão (SELEÇÃO) selecione a função desejada, ver a fig. 5.11. Cada vez que o botão SELEÇÃO for pressionado, um dos oito LEDs deverá acender para indicar a função escolhida; exemplo: 00000100 => LED 3 ligado, função 3 selecionada. Quando o botão de AJUSTE for solto, o sistema começa a funcionar conforme a função escolhida.

Desenvolva a programação por partes, unindo-as e testando com cuidado. Não esqueça: um bom programa é bem comentado e organizado!

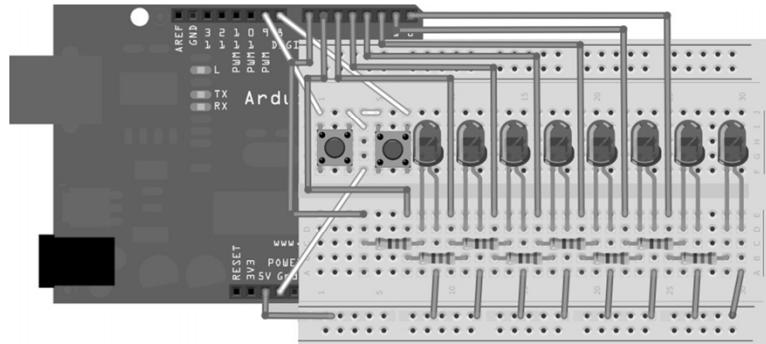
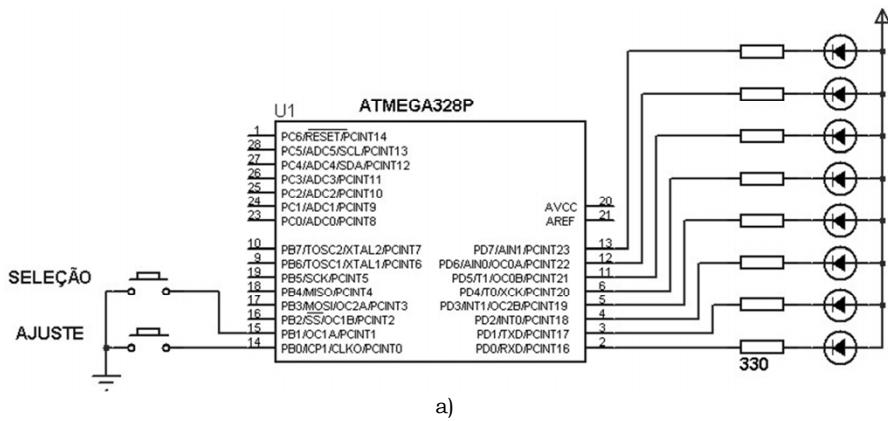


Fig. 5.11 – Sequencial de LEDs: a) esquemático e b) montagem no Arduino.

5.4 ACIONANDO DISPLAYS DE 7 SEGMENTOS

Um componente muito empregado no desenvolvimento de sistemas microcontrolados é o *display* de 7 segmentos. Esses *displays* geralmente são compostos por LEDs arranjados adequadamente em um encapsulamento, produzindo os dígitos numéricos que lhe são característicos. Na fig. 5.12, é apresentado o diagrama esquemático para uso de um *display* de anodo comum e os caracteres mais comuns produzidos por esse. Nos *displays* com catodo comum, o ponto comum dos LEDs é o terra e a tensão de alimentação deve ser aplicada individualmente em cada LED do *display*.

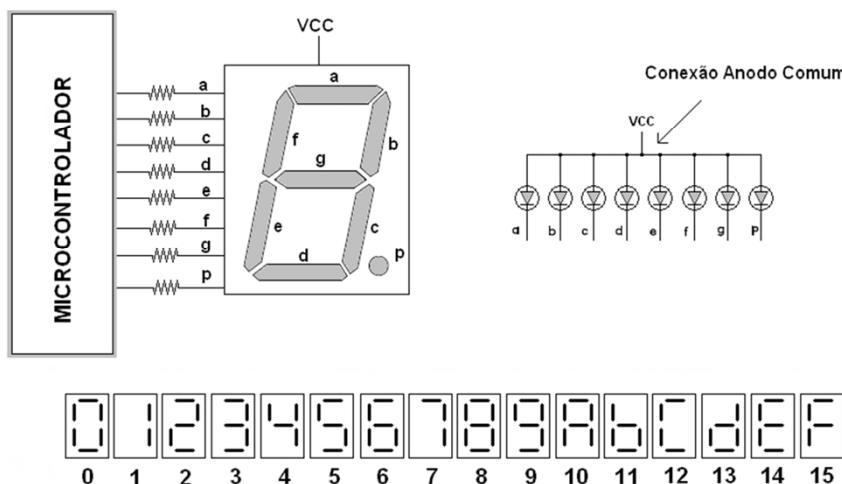


Fig. 5.12 – *Display* de 7 segmentos anodo comum.

Para o emprego de *displays*, é necessário decodificar o caractere que se deseja apresentar, ou seja, passá-lo da representação binária convencional para outra que represente corretamente o caractere no *display*, de acordo com o seu arranjo de LEDs. Na tab. 5.2, é apresentado o valor binário para os números hexadecimais de 0 até F, considerando o segmento *a* como sendo o bit menos significativo (LSB). Caso o *display* esteja ligado a um PORT de 8 bits, para ligar o ponto (p) basta habilitar o 8º bit (MSB).

Tab. 5.2 - Valores para a decodificação de *displays* de 7 segmentos.

Dígito	Anodo comum		Catodo comum	
	gfedcba		gfedcba	
0	0b1000000	0x40	0b0111111	0x3F
1	0b1111001	0x79	0b0000110	0x06
2	0b0100100	0x24	0b1011011	0x5B
3	0b0110000	0x30	0b1001111	0x4F
4	0b0011001	0x19	0b1100110	0x66
5	0b0010010	0x12	0b1101101	0x6D
6	0b0000010	0x02	0b1111101	0x7D
7	0b1111000	0x78	0b0000111	0x07
8	0b0000000	0x00	0b1111111	0x7F
9	0b0011000	0x18	0b1100111	0x67
A	0b0001000	0x08	0b1110111	0x77
B	0b0000011	0x03	0b1111100	0x7C
C	0b1000110	0x46	0b0111001	0x39
D	0b0100001	0x21	0b1011110	0x5E
E	0b0000110	0x06	0b1111001	0x79
F	0b0001110	0x0E	0b1110001	0x71

Na fig. 5.13, é apresentado um fluxograma para mostrar um número hexadecimal (de 0 até F) em um *display* de 7 segmentos quando um botão é pressionado. Se o botão é mantido pressionado, o valor é constantemente alterado e, após chegar ao valor F, retorna a 0. Os programas em *assembly* e C são apresentados na sequência. Em relação ao fluxograma, a diferença entre eles é que o programa em *assembly* fez uso de uma sub-rotina para decodificar o número e mostrá-lo no *display*. O circuito microcontrolado empregado encontra-se na fig. 5.14.

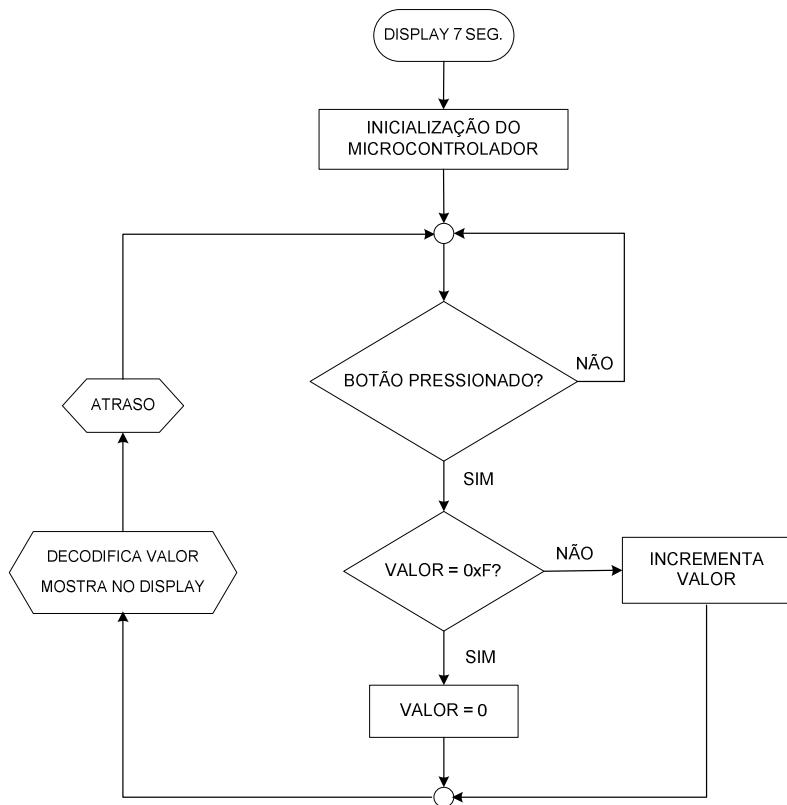


Fig. 5.13 – Fluxograma para apresentar um número hexadecimal de 0 até F quando um botão é pressionado.

Display_7Seg.asm

```

//=====
//      ESCRREVENDO EM UM DISPLAY DE 7 SEGMENTOS ANODO COMUM
//=====
//      Toda vez que um botão é pressionado o valor do Display muda(0->F)
//      mantendo-se o botão pressionado o incremento é automático
//=====
.equ BOTAO = PB0 //BOTAO é o substituto de PB0 na programação
.equ DISPLAY = PORTD //PORTD é onde está conectado o Display (seg a = LSB)
.def AUX = R16; //R16 tem agora o nome de AUX
//-----
.ORG 0x000

Inicializacoes:
    LDI AUX,0b11111110 //carrega AUX com o valor 0xFE (1 saída, 0 entrada)
    OUT DDRB,AUX        //configura PORTB, PB0 entrada e PB1 .. PB7 saídas
    LDI AUX,0xFF
    OUT PORTB,AUX       //habilita o pull-up do PB0 (demais pinos em 1)
    OUT DDRD,AUX        //PORTD como saída
    OUT PORTD,AUX       //desliga o display
  
```

```

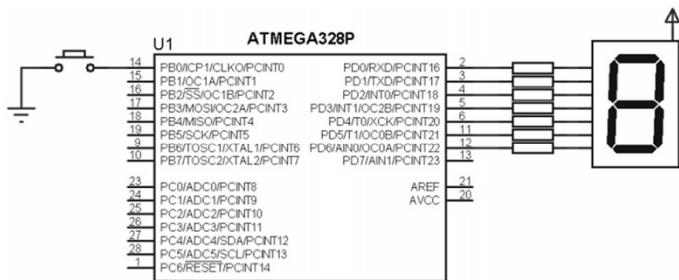
/*Para utilizar os pinos PD0 e PD1 como I/O genérico no Arduino é necessário
   desabilitar as funções TXD e RXD desses pinos*/
STS UCSR0B,R1  /*carrega o valor 0x00 (default de R1) em UCSR0B,
                  como ele esta na SRAM, usa-se STS*/
//-----
Principal:
SBIC PINB,BOTA0      //verifica se o botão foi pressionado, senão
RJMP Principal         //volta e fica preso no laço Principal
CPI AUX,0x0F           //compara se valor é máximo
BRNE Incr              //se não for igual, incrementa; senão, zera valor
LDI AUX,0x00
RJMP Decod

Incr:
INC AUX

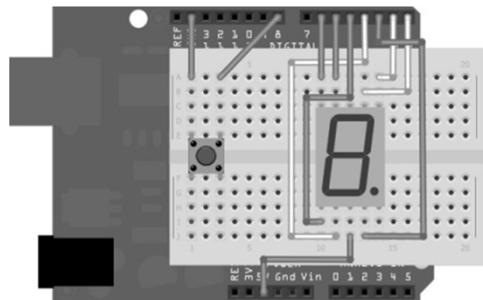
Decod:
RCALL Decodifica      //chama sub-rotina de decodificação
RCALL Atraso            /*incremento automático do display se o botão ficar
                           pressionado*/
RJMP Principal          //volta ler botão
//-----
//SUB-ROTTINA de atraso - Aprox. 0,2 s à 16 MHz
//-----
Atraso:
LDI R19,16              //repete os laços abaixo 16 vezes
volta:
DEC R17                 //decrementa R17
BRNE volta               //enquanto R17 > 0 fica decrementando R17
DEC R18                 //decrementa R18
BRNE volta               //enquanto R18 > 0 volta a decrementar R17
DEC R19                 //decrementa R19, começa com 0x02
BRNE volta
RET
//-----
//SUB-ROTTINA que decodifica um valor de 0 -> 15 para o display
//-----
Decodifica:
LDI ZH,HIGH(Tabela<<1) /*carrega o endereço da tabela no registrador Z, de
                           16 bits (trabalha como um ponteiro)*/
LDI ZL,LOW(Tabela<<1)    /*deslocando a esquerda todos os bits, pois o bit 0 é
                           para a seleção do byte alto ou baixo no end. de memória*/
ADD ZL,AUX                /*soma posição de memória correspondente ao nr. a
                           apresentar na parte baixa do endereço*/
BRCC le_tab                /*se houve Carry, incrementa parte alta do endereço,
                           senão lê diretamente a memória*/
INC ZH

le_tab:
LPM R0,Z                 //lê valor em R0
OUT DISPLAY,R0             //mostra no display
RET
//-----
// Tabela p/ decodificar o display: como cada endereço da memória flash é de 16 bits,
// acessa-se a parte baixa e alta na decodificação
//-----
Tabela: .dw 0x7940, 0x3024, 0x1219, 0x7802, 0x1800, 0x0308, 0x2146, 0x0E06
//           1 0     3 2     5 4     7 6     9 8     B A     D C     F E
//=====

```



a)



b)

Fig. 5.14 – Circuito para acionamento de um *display* de 7 segmentos anodo comum:
a) esquemático e b) montagem no Arduino.

Display_7Seg.c

```
===== //=====
//          ESCREVENDO EM UM DISPLAY DE 7 SEGMENTOS ANODO COMUM      //
===== //=====
#define F_CPU 16000000UL //define a frequência do microcontrolador em 16MHz

#include <avr/io.h>      //definições do componente especificado
#include <util/delay.h>   //biblioteca para o uso das rotinas de _delay_
#include <avr/pgmspace.h> //biblioteca para poder gravar dados na memória flash

//Definições de macros - para o trabalho com os bits de uma variável
#define tst_bit(Y,bit_x) (Y&(1<<bit_x)) //testa o bit x da variável Y (retorna 0 ou 1)

#define DISPLAY PORTD    //define um nome auxiliar para o display
#define BOTAO  PB0        //define PB0 com o nome de BOTAO

//variável gravada na memória flash
const unsigned char Tabela[] PROGMEM = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12,
                                         0x02, 0x78, 0x00, 0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E};
//-----
```

```

int main()
{
    unsigned char valor = 0; //declara variável local

    DDRB = 0b11111110;    //PB0 como pino de entrada, os demais pinos como saída
    PORTB= 0x01;          //habilita o pull-up do PB0
    DDRD = 0xFF;          //PORTD como saída (display)
    PORTD= 0xFF;          //desliga o display
    UCSR0B = 0x00;         //PD0 e PD1 como I/O genérico, para uso no Arduino

    while(1)                  //laço infinito
    {
        if(!tst_bit(PINB,BOTA0)) //se o botão for pressionado executa
        {
            if(valor==0xF)      //se o valor for igual a 0xF, zera o valor,
                valor=0;
            else                 //se não o incrementa
                valor++;

            //decodifica o valor e mostra no display, busca o valor na Tabela.
            DISPLAY = pgm_read_byte(&Tabela[valor]);

            _delay_ms(200); //atraso para incremento automático do nr. no display

        }//if botão
    } //laço infinito
}
//=====

```

O emprego de tabelas é muito usual e poderoso na programação de microcontroladores. Elas devem ser armazenadas na memória de programa para evitar o desperdício da memória RAM e da própria memória *flash*. A gravação de dados na memória RAM aumenta o tamanho do código porque a movimentação das constantes para as posições da RAM é realizada pelo programa. Isso significa que é duplamente prejudicial empregar a RAM para armazenar dados que não serão alterados durante o programa. Esse problema pode passar despercebido quando se programa em linguagem C. Com o compilador AVR-GCC, a gravação de dados na memória *flash* é feita com uso da biblioteca **pgmspace.h** e do comando **const ... PROGMEM** na declaração da variável, conforme apresentado no programa anterior.

É importante salientar que cada posição de memória *flash* do AVR ocupa 16 bits. Entretanto, o hardware permite o acesso a dados gravados por bytes individualmente. O bit 0 do registrador ZL informa se deve ser lido o byte baixo ou alto do endereço. Para isso, é preciso concatenar 2 bytes para cada posição de memória (ver a tabela do programa *assembly*

apresentado anteriormente). Em C, a programação é bem mais fácil e detalhes do microcontrolador não precisam ser conhecidos, como exigido ao se programar em *assembly*.

Exercícios:

5.13 – Elaborar um programa para apresentar em um *display* de 7 segmentos um número aleatório²⁵ entre 1 e 6 quando um botão for pressionado, ou seja, crie um dado eletrônico. Empregue o mesmo circuito da fig. 5.14.

5.14 – Elaborar um programa para apresentar nos LEDs da fig. 5.15 um número aleatório entre 1 e 6, formando os números de um dado (mesma lógica do exercício acima).

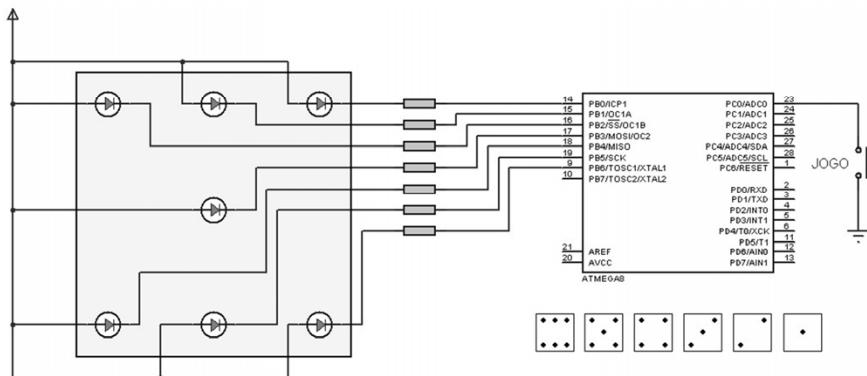


Fig. 5.15 – Dado eletrônico com LEDs.

Obs.: a pinagem do ATmega8 é igual a do ATmega328.

²⁵ Na verdade, criar um número puramente aleatório é difícil, o mais fácil é um pseudoaleatório. Neste exercício, o objetivo é não empregar as bibliotecas padrão do C. A ideia é utilizar o botão para gerar o evento de sorteio do número. Dessa forma, um contador pode ficar contando continuamente de 1 até 6 e, quando o botão for pressionado, um número da contagem será selecionado.

5.5 ACIONANDO LCDs 16 x 2

Os módulos LCDs são interfaces de saída muito úteis em sistemas microcontrolados. Estes módulos podem ser gráficos ou a caractere (alfanuméricicos). Os LCDs comuns, tipo caractere, são especificados em número de linhas por colunas, sendo mais usuais as apresentações 16×2, 16×1, 20×2, 20×4, 8×2. Além disso, os módulos podem ser encontrados com *backlight* (LEDs para iluminação de fundo), facilitando a leitura em ambientes escuros. Os LCDs mais comuns empregam o CI controlador HD44780 da Hitachi com interface paralela. Há no mercado também LCDs com controle serial, sendo que novos LCDs são constantemente criados.

A seguir, será descrito o trabalho com o LCD de 16 caracteres por 2 linhas (ver o apêndice B); o uso de qualquer outro baseado no controlador HD44780 é similar. Na fig. 5.16, é apresentado o circuito microcontrolado com um *display* de 16×2.

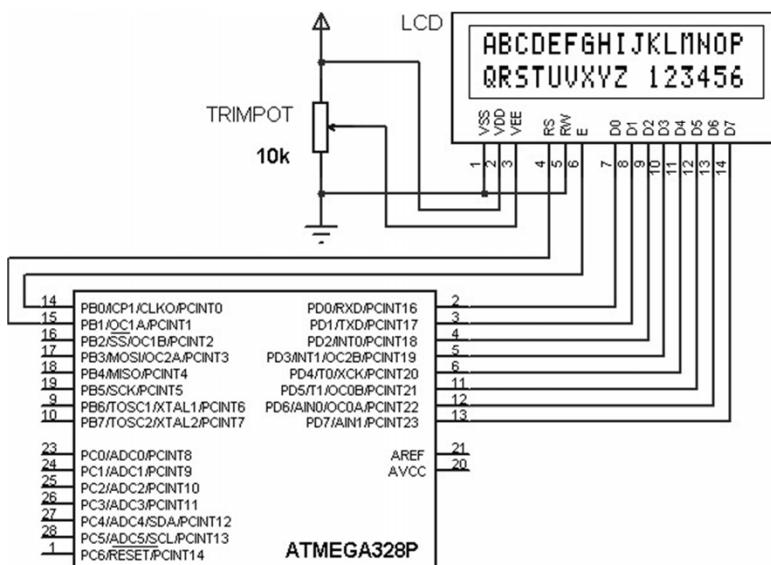


Fig. 5.16 – Circuito para acionamento de um LCD 16×2 usando 8 vias de dados.

Existem duas possibilidades de comunicação com o *display* da fig. 5.16. Uma é empregando 8 via de dados para a comunicação (D0-D7) e a outra, 4 vias de dados (D4-D7). Nesta última, o dado é enviado separadamente em duas partes (2 *nibbles*).

5.5.1 INTERFACE DE DADOS DE 8 BITS

Para ler ou escrever no *display* LCD com uma via de dados de 8 bits, é necessário a seguinte sequência de comandos:

1. Levar o pino R/W (*Read/Write*) para 0 lógico se a operação for de escrita e 1 lógico se for de leitura. Aterra-se esse pino se não há necessidade de monitorar a resposta do LCD (forma mais usual de trabalho).
2. Levar o pino RS (*Register Select*) para o nível lógico 0 ou 1 (instrução ou caractere).
3. Transferir os dados para a via de dados (8 bits).
4. Gerar um pulso de habilitação. Ou seja, levar o pino E (*Enable*) para 1 lógico e, após um pequeno tempo, para 0 lógico.
5. Empregar uma rotina de atraso entre as instruções ou fazer a leitura do *busy flag* (o bit 7 da linha de dados que indica que o *display* está ocupado) antes do envio da instrução, enviando-a somente quando esse *flag* for 0 lógico.

Os passos 1, 2 e 3 podem ser efetuados em qualquer sequência, pois o pulso de habilitação é que faz o controlador do LCD ler os dados dos seus pinos. É importante respeitar os tempos de resposta do LCD à transição dos sinais enviados ao mesmo²⁶.

Toda vez que a alimentação do LCD é ligada, deve ser executada uma rotina de inicialização. O LCD começa a responder aproximadamente 15 ms após a tensão de alimentação atingir 4,5 V. Como não se conhece o

²⁶ Para uma melhor compreensão sobre o assunto, consultar o manual do fabricante do LCD empregado.

tempo necessário para que ocorra a estabilização da tensão no circuito onde está colocado o LCD, pode ser necessário frações bem maiores de tempo para que o LCD possa responder a comandos. Muitas vezes, esse detalhe é esquecido e o LCD não funciona adequadamente. Para corrigir esse problema, basta colocar uma rotina de atraso suficientemente grande na inicialização do LCD. Na fig. 5.17, é apresentado o fluxograma de inicialização do LCD conforme especificação da Hitachi. Se desejado, o *busy flag* pode ser lido após o ajuste do modo de utilização do *display*. Os comandos para o LCD são detalhados no apêndice B.

A seguir são apresentadas as rotinas de escrita no LCD em conjunto com um programa exemplo que escreve na linha superior do LCD “ABCDEFGHIJKLMNP” e “QRSTUWXYZ 123456” na linha inferior (circuito da fig. 5.16). Muitos programadores não utilizam a rotina de inicialização completa do diagrama da fig. 5.17, respeitando apenas o tempo de resposta do *display*, seguido do modo de utilização e dos outros controles, prática que geralmente funciona.

Para a visualização dos caracteres é imprescindível o emprego do *trimpot* (fig. 5.16) para ajuste do contraste do *display* de cristal líquido.

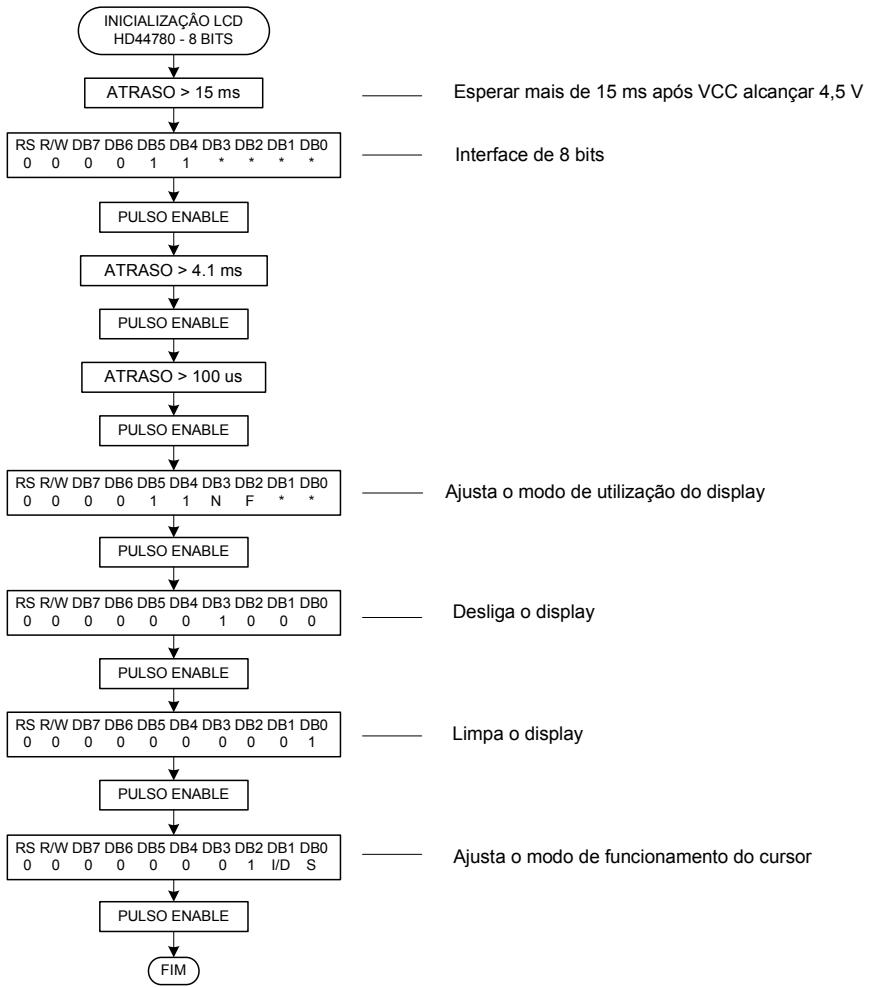


Fig. 5.17 – Rotina de inicialização de 8 bits para um LCD com base no CI HD44780.

LCD_8bits.c

```
//===================================================================== //
//      ACIONANDO UM DISPLAY DE CRISTAL LIQUIDO DE 16x2           //
//                                                               //
//          Interface de dados de 8 bits                         //
//                                                               //
//===================================================================== //
#define F_CPU 16000000UL //define a frequência do microcontrolador - 16MHz

#include <avr/io.h>      //definições do componente especificado
#include <util/delay.h>  //biblioteca para o uso das rotinas de delay
#include <avr/pgmspace.h> //uso de funções para salvar dados na memória de programa

//Definições de macros - empregadas para o trabalho com o bits
#define set_bit(Y,bit_x) (Y|=(1<<bit_x)) //ativa o bit x da variável Y
#define clr_bit(Y,bit_x) (Y&~(1<<bit_x)) //limpa o bit x da variável Y
#define tst_bit(Y,bit_x) (Y&(1<<bit_x)) //testa o bit x da variável Y
#define cpl_bit(Y,bit_x) (Y^=(1<<bit_x)) //troca o estado do bit x da variável Y

//para uso no LCD (deve estar na mesma linha)
#define pulso_enable() _delay_us(1); set_bit(CONTR_LCD,E); _delay_us(1);
                                         clr_bit(CONTR_LCD,E); _delay_us(45)

#define DADOS_LCD    PORTD    //8 bits de dados do LCD na porta D
#define CONTR_LCD    PORTB    //os pinos de controle estão no PORTB
#define RS           PB1      //pino de instrução ou dado para o LCD
#define E            PB0      //pino de enable do LCD

//mensagem armazenada na memória flash
const unsigned char msg1[] PROGMEM = "ABCDEFGHIJKLMNP";

//-----
//Sub-rotina para enviar caracteres e comandos ao LCD
//-----
void cmd_LCD(unsigned char c, char cd)//c é o dado e cd indica se é instrução ou caractere
{
    DADOS_LCD = c;

    if(cd==0)
        clr_bit(CONTR_LCD,RS); //RS = 0
    else
        set_bit(CONTR_LCD,RS); //RS = 1

    pulso_enable();

    //se for instrução de limpeza ou retorno de cursor espera o tempo necessário
    if((cd==0) && (c<4))
        _delay_ms(2);
}

//-----
//Sub-rotina de inicialização do LCD
//-----
void inic_LCD_8bits()//sequência ditada pelo fabricante do circuito de controle do LCD
{
    clr_bit(CONTR_LCD,RS);//o LCD será só escrito então R/W é sempre zero

    _delay_ms(15); /*tempo para estabilizar a tensão do LCD, após VCC ultrapassar
                    4.5 V (pode ser bem maior na prática)*/

    DADOS_LCD = 0x38; //interface 8 bits, 2 linhas, matriz 7x5 pontos

    pulso_enable(); //enable respeitando os tempos de resposta do LCD
    _delay_ms(5);
    pulso_enable();
    _delay_us(200);
```

```

pulso_enable();
pulso_enable();

cmd_LCD(0x08,0);      //desliga LCD
cmd_LCD(0x01,0);      //limpa todo o display
cmd_LCD(0xC0,0);      //mensagem aparente cursor inativo não piscando
cmd_LCD(0x80,0);      //escreve na primeira posição a esquerda - 1ª linha
}
//-----
//Sub-rotina de escrita no LCD
//-----
void escreve_LCD(char *c)
{
    for ( ; *c!=0;c++) cmd_LCD(*c,1);
}
//-----
int main()
{
    unsigned char i;

    DDRB = 0xFF;      //PORTB como saída
    DDRD = 0xFF;      //PORTD como saída
    UCSR0B = 0x00;    //habilita os pinos PD0 e PD1 como I/O para uso no Arduino

    inic_LCD_8bits(); //inicializa o LCD

    for(i=0;i<16;i++) //enviando caractere por caractere
        cmd_LCD(pgm_read_byte(&msg1[i]),1); //lê na memória flash e usa cmd_LCD

    cmd_LCD(0xC0,0); //desloca o cursor para a segunda linha do LCD
    escreve_LCD("QRSTUVXYZ 123456");//a cadeia de caracteres é criada na RAM

    for(;;);           //laço infinito
}
//=====

```

5.5.2 INTERFACE DE DADOS DE 4 BITS

Nesta seção, o trabalho com o LCD e suas rotinas serão melhores detalhados, visto que utilizar uma interface de dados de 8 bits para um LCD de 16×2 não é recomendado. Isso se deve ao uso excessivo de vias para o acionamento do *display* (10 ou 11). O emprego de 4 bits de dados libera 4 pinos de I/O do microcontrolador para outras atividades, além de diminuir o número de trilhas necessárias na placa de circuito impresso. O custo é um pequeno aumento na complexidade do programa de controle do LCD, o que consome alguns bytes a mais de programação.

Para ler ou escrever no *display* LCD com uma via de dados de 4 bits, é necessário a seguinte sequência de comandos:

1. Levar o pino R/W (*Read/ Write*) para 0 lógico se a operação for de escrita e 1 lógico se for de leitura. Aterra-se esse pino se não há necessidade de monitorar a resposta do LCD (forma mais usual de trabalho).
2. Levar o pino RS (*Register Select*) para o nível lógico 0 ou 1 (instrução ou caractere).
3. Transferir a parte mais significativa dos dados para a via de dados (4 bits mais significativos (MSB) – *nibble* maior).
4. Gerar um pulso de habilitação. Ou seja, levar o pino E (*Enable*) para 1 lógico e após um pequeno tempo de espera para 0 lógico.
5. Transferir a parte menos significativa dos dados para a via de dados (4 bits menos significativos (LSB) – *nibble* menor).
6. Gerar outro pulso de habilitação.
7. Empregar uma rotina de atraso entre as instruções ou fazer a leitura do *busy flag* (o bit 7 da linha de dados que indica que o *display* está ocupado) antes do envio da instrução, enviando-a somente quando esse *flag* for 0 lógico.

Os passos 1, 2 e 3 podem ser efetuados em qualquer sequência, pois o pulso de habilitação é que faz o controlador do LCD ler os dados dos seus pinos.

Na fig. 5.18, é apresentado o circuito microcontrolado com o LCD 16×2 com via de dados de 4 bits (conexão igual a do módulo LCD *shield* da Ekitszone²⁷). Na sequência, mostram-se o fluxograma de inicialização do *display*, de acordo com a Hitachi, e o programa de controle do LCD; os arquivos com os programas para o trabalho com o LCD foram organizados de forma estruturada (ver a seção 4.5.11). O resultado prático é visto na fig. 5.20.

²⁷ www.ekitszone.com. Existem outros módulos LCD disponíveis no mercado, depende somente da escolha do usuário. Se desejado o circuito pode ser montado em uma matriz de contatos.

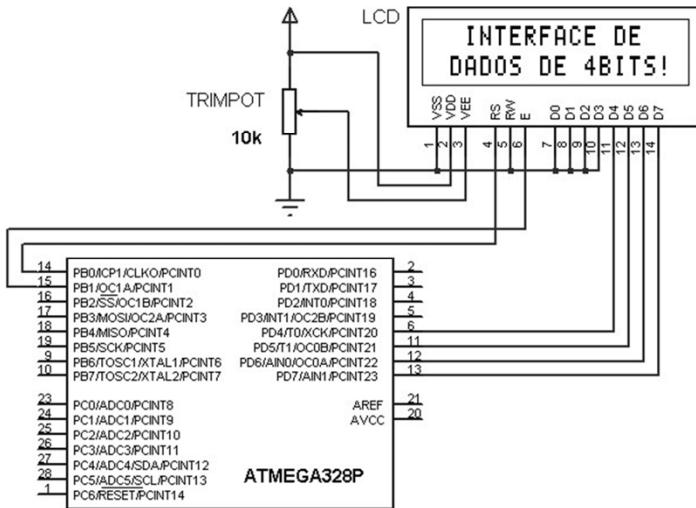


Fig. 5.18 – Circuito para acionamento de um LCD 16 × 2 com interface de dados de 4 bits.

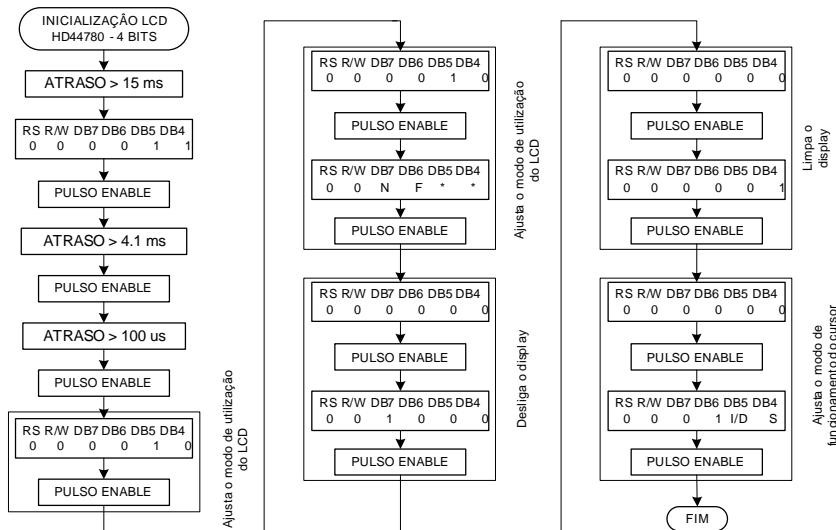


Fig. 5.19 – Rotina de inicialização de 4 bits para um LCD com base no CI HD44780.

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL //define a frequência do microcontrolador - 16MHz

#include <avr/io.h> //definições do componente especificado
#include <util/delay.h> //biblioteca para o uso das rotinas de _delay_ms e _delay_us()
#include <avr/pgmspace.h> //para a gravação de dados na memória flash

//Definições de macros para o trabalho com bits
#define set_bit(y,bit) (y|=(1<<bit)) //coloca em 1 o bit x da variável Y
#define clr_bit(y,bit) (y&~(1<<bit)) //coloca em 0 o bit x da variável Y
#define cpl_bit(y,bit) (y^=(1<<bit)) //troca o estado lógico do bit x da variável Y
#define tst_bit(y,bit) (y&(1<<bit)) //retorna 0 ou 1 conforme leitura do bit

#endif
```

LCD_4bits.c (programa principal)

```
//=====
//      ACIONANDO UM DISPLAY DE CRISTAL LIQUIDO DE 16x2                      //
//                                         Interface de dados de 4 bits           //
//=====

#include "def_principais.h" //inclusão do arquivo com as principais definições
#include "LCD.h"

//definição para acessar a memória flash
prog_char mensagem[] = " DADOS DE 4BITS!\0"; //mensagem armazenada na memória flash

//-----
int main()
{
    DDRD = 0xFF;                  //PORTD como saída
    DDRB = 0xFF;                  //PORTB como saída

    inic_LCD_4bits();             //inicializa o LCD
    escreve_LCD(" INTERFACE DE"); //string armazenada na RAM
    cmd_LCD(0xC0,0);              //desloca cursor para a segunda linha
    escreve_LCD_Flash(mensagem);  //string armazenada na flash

    for(;;){}                     //laço infinito, aqui vai o código principal
}
//=====
```

LCD.h (arquivo de cabeçalho do LCD.c)

```
#ifndef _LCD_H
#define _LCD_H

#include "def_principais.h"

#define DADOS_LCD    PORTD//4 bits de dados do LCD no PORTD
#define nibble_dados 1 /*0 para via de dados do LCD nos 4 LSBs do PORT
                     empregado (Px0-D4, Px1-D5, Px2-D6, Px3-D7), 1 para via de
                     dados do LCD nos 4 MSBs do PORT empregado (Px4-D4, Px5-D5,
                     Px6-D6, Px7-D7) */

#define CONTR_LCD    PORTB//PORT com os pinos de controle do LCD (pino R/W em 0).
#define E            PB1 //pino de habilitação do LCD (enable)
#define RS           PB0 //pino para informar se o dado é uma instrução ou caractere

#define tam_vetor   5 //número de dígitos individuais para a conversão por ident_num()
#define conv_ascii   48 /*48 se ident_num() deve retornar um número no formato ASCII (0 para
                     formato normal)*/

//sinal de habilitação para o LCD
#define pulso_enable() _delay_us(1); set_bit(CONTR_LCD,E); _delay_us(1);
                                         clr_bit(CONTR_LCD,E); _delay_us(45)

//protótipo das funções
void cmd_LCD(unsigned char c, char cd);
void inic_LCD_4bits();
void escreve_LCD(char *c);
void escreve_LCD_Flash(const char *c);

void ident_num(unsigned int valor, unsigned char *disp);

#endif
```

LCD.c (funções para o LCD)

```
===== //
// Sub-rotinas para o trabalho com um LCD 16x2 com via de dados de 4 bits      //
// Controlador HD44780 - Pino R/W aterrado                                     //
// A via de dados do LCD deve ser ligado aos 4 bits mais significativos ou   //
// aos 4 bits menos significativos do PORT do uC                                //
===== //

#include "LCD.h"

//-----
// Sub-rotina para enviar caracteres e comandos ao LCD com via de dados de 4 bits
//-----
//c é o dado e cd indica se é instrução ou caractere (0 ou 1)
void cmd_LCD(unsigned char c, char cd)
{
    if(cd==0)                      //instrução
        clr_bit(CONTR_LCD,RS);
    else                            //caractere
        set_bit(CONTR_LCD,RS);

    //primeiro nibble de dados - 4 MSB
    #if (nibble_dados)//compila o código para os pinos de dados do LCD nos 4 MSB do PORT
        DADOS_LCD = (DADOS_LCD & 0x0F)|(0xF0 & c);
    #else                  //compila o código para os pinos de dados do LCD nos 4 LSB do PORT
        DADOS_LCD = (DADOS_LCD & 0x0F)|(c>>4);
    #endif

    pulso_enable();
```

```

//segundo nibble de dados - 4 LSB
#if (nibble_dados) //compila o código para os pinos de dados do LCD nos 4 MSB do PORT
    DADOS_LCD = (DADOS_LCD & 0x0F) | (0xF0 & (c<<4));
#else //compila o código para os pinos de dados do LCD nos 4 LSB do PORT
    DADOS_LCD = (DADOS_LCD & 0xF0) | (0x0F & c);
#endif

pulso_enable();

if((cd==0) && (c<4)) //se for instrução de retorno ou limpeza espera LCD estar pronto
    _delay_ms(2);
}

//-----
//Sub-rotina para inicialização do LCD com via de dados de 4 bits
//-----
void inic_LCD_4bits()//sequência ditada pelo fabricante do circuito integrado HD44780
{
    //o LCD será só escrito. Então, R/W é sempre zero.

    clr_bit(CONTR_LCD,RS); //RS em zero indicando que o dado para o LCD será uma instrução
    clr_bit(CONTR_LCD,E); //pino de habilitação em zero

    _delay_ms(20); /*tempo para estabilizar a tensão do LCD, após VCC
                    ultrapassar 4.5 V (na prática pode ser maior).*/

    //interface de 8 bits
    #if (nibble_dados)
        DADOS_LCD = (DADOS_LCD & 0x0F) | 0x30;
    #else
        DADOS_LCD = (DADOS_LCD & 0xF0) | 0x03;
    #endif

    pulso_enable(); //habilitação respeitando os tempos de resposta do LCD
    _delay_ms(5);
    pulso_enable();
    _delay_us(200);
    pulso_enable(); //até aqui ainda é uma interface de 8 bits.

    //interface de 4 bits, deve ser enviado duas vezes (a outra está abaixo)
    #if (nibble_dados)
        DADOS_LCD = (DADOS_LCD & 0x0F) | 0x20;
    #else
        DADOS_LCD = (DADOS_LCD & 0xF0) | 0x02;
    #endif

    pulso_enable();
    cmd_LCD(0x28,0); //interface de 4 bits 2 linhas (aqui se habilita as 2 linhas)
    //só são enviados os 2 nibbles (0x2 e 0x8)
    cmd_LCD(0x08,0); //desliga o display
    cmd_LCD(0x01,0); //limpa todo o display
    cmd_LCD(0x0C,0); //mensagem aparente cursor inativo não piscando
    cmd_LCD(0x80,0); //inicializa cursor na primeira posição a esquerda - 1a linha
}

//-----
//Sub-rotina de escrita no LCD - dados armazenados na RAM
//-----
void escreve_LCD(char *c)
{
    for (; *c!=0;c++) cmd_LCD(*c,1);
}

//-----
//Sub-rotina de escrita no LCD - dados armazenados na FLASH
//-----
void escreve_LCD_Flash(const char *c)
{
    for (;pgm_read_byte(&(*c))!=0;c++) cmd_LCD(pgm_read_byte(&(*c)),1);
}

```

```

//-----
//Conversão de um número em seus dígitos individuais - função auxiliar
//-----
void ident_num(unsigned int valor, unsigned char *disp)
{
    unsigned char n;

    for(n=0; n<tam_vetor; n++)
        disp[n] = 0 + conv_ascii;      //limpa vetor para armazenagem dos dígitos

    do
    {
        *disp = (valor%10) + conv_ascii; //pega o resto da divisão por 10
        valor /=10;                   //pega o inteiro da divisão por 10
        disp++;
    }

    }while (valor!=0);
}
//-----

```

É fundamental compreender as funções apresentadas para a programação do LCD. As funções foram desenvolvidas para serem flexíveis quanto à conexão dos pinos do LCD ao microcontrolador, a única ressalva é quanto a disposição dos 4 pinos de dados do LCD (D4-D7), os quais devem ser conectados em um *nibble* alto ou em um *nibble* baixo do PORT utilizado. As definições dos pinos é feita no arquivo LCD.h.



Fig. 5.20 – Resultado do programa para controle de um LCD 16 × 2 com interface de dados de 4 bits (módulo LCD – *shield*, da Ekitszone).

A principal função para o controle do LCD é a **cmd_LCD(dado, 0 ou 1)**, que recebe dois parâmetros: o dado que se deseja enviar ao LCD e o

número 0 ou 1; onde 0 indica que o dado é uma instrução e 1 indica que o dado é um caractere.

A função **inic_LCD_4bits()** deve ser utilizada no início do programa principal para a correta inicialização do LCD. Existe uma sequência de comandos que deve ser seguida para que o LCD possa funcionar corretamente.

A função **escreve_LCD("frase")** recebe uma *string*, ou seja um conjunto de caracteres. Como na programação em C toda *string* é finalizada com o caractere nulo (0), essa função se vale desse artifício para verificar o final da *string*. Deve-se ter cuidado ao se utilizar essa função, porque a *string* é armazenada na memória RAM do microcontrolador, o que pode limitar a memória disponível para o programa. Para evitar esse problema, pode-se utilizar a função **escreve_LCD_Flash(frase)**, onde a frase, previamente declarada no programa, é armazenada na memória *flash*.

Uma vez inicializado o LCD, se escolhe em qual posição dele se deseja escrever. Cada vez que um caractere é escrito, o cursor é automaticamente deslocado para a próxima posição de escrita, à direita ou à esquerda conforme inicialização (ver a tabela B.3 do apêndice B). Assim, é importante entender como mudar a posição do cursor antes de escrever o caractere. Na fig. 5.21, são apresentados os endereços correspondentes a cada caractere do LCD 16 × 2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Linha 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
Linha 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF

Fig. 5.21 – Endereços para escrita num LCD 16 × 2.

Por exemplo, quando se deseja escrever o caractere A na 6^a posição da linha superior (linha 1), deve ser empregado o seguinte código:

```
cmd_LCD(0x85,0);      //desloca cursor para o endereço 0x86  
cmd_LCD('A',1);       //escrita do caractere
```

Para escrever o conjunto de caracteres “Alo mundo!” na linha inferior começando na terceira posição, deve-se empregar o código:

```
cmd_LCD(0xC2,0);           //desloca cursor para o endereço 0xC2  
escreve_LCD("Alo mundo!"); //escrita da string
```

A mensagem não aparecerá caso se escreva em uma posição que não existe na tela do LCD. Se outro LCD for empregado, como por exemplo um 20×4 (20 caracteres por 4 linhas) o endereçamento será diferente. Na fig. 5.22, são apresentados os endereços dos caracteres para um LCD 20×4 . A linha 3 é continuação da linha 1 e a linha 4, continuação da linha 2. Na dúvida, o manual do fabricante deve ser consultado.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Linha 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90	91	92	93
Linha 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3
Linha 3	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F	A0	A1	A2	A3	A4	A5	A6	A7
Linha 4	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF	E0	E1	E2	E3	E4	E5	E6	E7

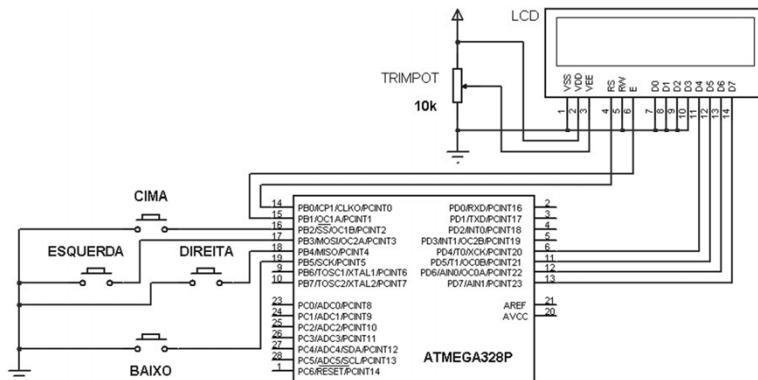
Fig. 5.22 – Endereços para a escrita num LCD 20×4 .

Exercícios:

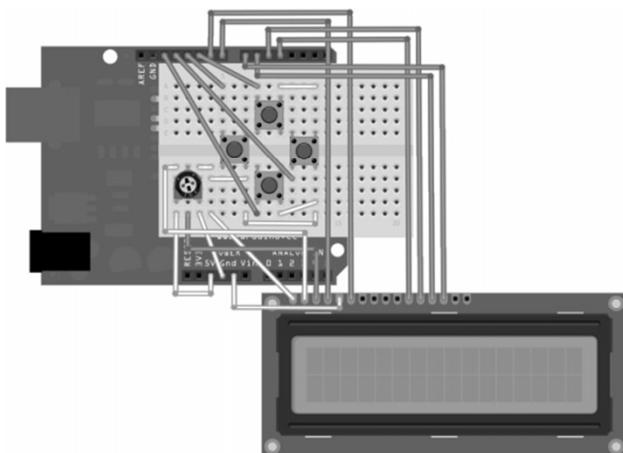
5.15 – Elaborar um programa para deslocar um caractere ‘*’ (asterisco) no LCD da fig. 5.18, da esquerda para a direita, ao chegar ao final da linha o caractere deve retornar (vai e vem).

5.16 – Repetir o exercício 5.15 empregando as duas linhas do LCD. Ao chegar ao final da linha superior, o asterisco começa na linha inferior (endereço 0xD3). Dessa forma, na linha superior o asterisco se desloca da esquerda para a direita e na linha inferior, da direita para a esquerda.

5.17 – Elaborar um programa para realizar o movimento de um cursor num LCD 16×2 com o uso de 4 botões, conforme fig. 5.23.



a)



b)

Fig. 5.23 – Exercício 5.17: a) esquemático e b) montagem no Arduino.

5.5.3 CRIANDO NOVOS CARACTERES

Os códigos dos caracteres recebidos pelo LCD são armazenados em uma RAM chamada DDRAM (*Data Display RAM*), transformados em um caractere no padrão matriz de pontos e apresentados na tela do LCD (ver a tabela no apêndice B para o conjunto de caracteres do HD44780). Para produzir os caracteres nesse padrão, o módulo LCD incorpora uma CGROM (*Character Generator ROM*). Os LCDs também possuem uma

CGRAM (*Character Generator RAM*) para a gravação de caracteres especiais. Oito caracteres programáveis podem ser escritos na CGRAM e apresentam os códigos fixos 0x00 a 0x07. Os dados da CGRAM são apresentados em um mapa de bits de 8 bytes, dos quais se utilizam 7, com 5 bits cada, sendo 1 byte reservado para o cursor. Dessa forma, qualquer caractere é representado por uma matriz de pontos 7×5 . Na fig. 5.24, apresenta-se o mapa de bits para o símbolo Δ .

Endereço da CGRAM	Mapa de bits	Dado
0x48		0b00100
0x49		0b00100
0x4A		0b01010
0x4B		0b01010
0x4C		0b10001
0x4E		0b11111
0x4F		0b00000

Fig. 5.24 – Gravação do símbolo Δ na CGRAM, matriz 5×7 . Esse caractere será selecionado pelo código 0x01.

Como há espaço para 8 caracteres, estão disponíveis 64 bytes na CGRAM. Na tab. 5.3, são apresentados os endereços base onde devem ser criados os novos caracteres e os seus respectivos códigos para escrita no *display*. Na fig. 5.24, por exemplo, o símbolo Δ será selecionado pelo código 0x01, pois o mesmo possui o endereço base 0x48 da CGRAM.

Tab. 5.3 – Endereço para criação de caracteres novos e seus códigos de chamada.

Endereço base	0x40	0x48	0x50	0x58	0x60	0x68	0x70	0x78
Código do caractere	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07

A seguir, apresenta-se o programa que cria dois caracteres novos, um no endereço 0x40 e o outro no 0x48. Após criados, os mesmos possuem o códigos 0x00 e 0x01, respectivamente (ver a tabela 5.3). Na fig. 5.25, apresenta-se o resultado para o LCD 16×2 .

LCD_4bits_2new_caract.c

```
//===================================================================== //
//          CRIANDO CARACTERES PARA O LCD 16x2                      //
//          Via de dados de 4 bits                                     //
//===================================================================== //

#include "def_principais.h"//inclusão do arquivo com as principais definições
#include "LCD.h"

//informação para criar caracteres novos armazenada na memória flash
const unsigned char carac1[] PROGMEM = {0b01110,//C
                                         0b10001,
                                         0b10000,
                                         0b10000,
                                         0b10101,
                                         0b01110,
                                         0b10000};

const unsigned char carac2[] PROGMEM = {0b00100,//Delta
                                         0b00100,
                                         0b01010,
                                         0b01010,
                                         0b10001,
                                         0b11111,
                                         0b00000};

//-----------------------------------------------------------------------------------------------------------------
int main()
{
    unsigned char k;

    DDRD = 0xFF;           //PORTD como saída
    DDRB = 0xFF;           //PORTB como saída

    inic_LCD_4bits();      //inicializa o LCD

    cmd_LCD(0x40,0);      //endereço base para gravar novo segmento
    for(k=0;k<7;k++)       //grava 8 bytes na DDRAM começando no end. 0x40
        cmd_LCD(pgm_read_byte(&carac1[k]),1);
    cmd_LCD(0x00,1);       /*apaga última posição do end. da CGRAM para evitar algum
                           dado espúrio*/

    cmd_LCD(0x48,0);      //endereço base para gravar novo segmento
    for(k=0;k<7;k++)       //grava 8 bytes na DDRAM começando no end. 0x48
        cmd_LCD(pgm_read_byte(&carac2[k]),1);
    cmd_LCD(0x00,1);       /*apaga última posição do end. da CGRAM para evitar algum
                           dado espúrio*/
    cmd_LCD(0x80,0);      //endereço a posição para escrita dos caracteres
    cmd_LCD(0x00,1);       //apresenta primeiro caractere 0x00
    cmd_LCD(0x01,1);       //apresenta segundo caractere 0x01

    for();};               //laço infinito
}
```

Para usar a diretiva PROGMEM e poder gravar dados na memória flash, é necessário incluir a biblioteca **pgmspace.h** no arquivo **def_principais.h** apresentado anteriormente: #include <avr/pgmspace.h>



Fig. 5.25 - Dois caracteres novos: C e Δ.

O código citado anteriormente apresenta uma forma de criar caracteres individualmente. Em uma programação mais eficiente, uma única matriz de informação deve ser utilizada.

5.5.4 NÚMEROS GRANDES

Utilizar 4 caracteres do LCD 16×2 para criar números grandes é uma técnica interessante de escrita. Ela pode ser muito útil, por exemplo, para a representação de horas. Com um pouco de raciocínio, utilizando os caracteres disponíveis (tab. B.4 do apêndice B) e criando os 8 permitidos, é possível montar os dígitos de 0 até 9 empregando 4 caracteres para cada número.

Na fig. 5.26, são apresentados os 8 caracteres a serem criados e seus respectivos códigos, os quais, quando organizados adequadamente em conjunto com os caracteres existentes para o LCD, permitem formar os números grandes, conforme fig. 5.27.

Matriz de pontos	Caracteres criados
7 5	 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07

Código dos caracteres criados

Fig. 5.26 – Caracteres criados para se poder escrever números grandes num LCD 16×2 .



Fig. 5.27 – Organização de 4 caracteres para formar números grandes em um LCD 16×2.

Para mostrar um número no *display*, é necessário escrever dois caracteres na linha superior do LCD e os outros dois na linha inferior, de acordo com os caracteres que formam cada número. A seguir, é apresentada uma matriz com os códigos para a formação dos números grandes conforme os códigos dos caracteres da fig. 5.25 e os disponíveis na tab. B.4 do apêndice B. Cada linha da matriz representa os 4 caracteres necessários para gerar o número.

```
//=====
unsigned char Nr_Grande[10][4] = {{0x01, 0x02, 0x4C, 0x00}, //nr. 0
                                  {0x20, 0x7C, 0x20, 0x7C}, //nr. 1
                                  {0x04, 0x05, 0x4C, 0x5F}, //nr. 2
                                  {0x06, 0x05, 0x5F, 0x00}, //nr. 3
                                  {0x4C, 0x00, 0x20, 0x03}, //nr. 4
                                  {0x07, 0x04, 0x5F, 0x00}, //nr. 5
                                  {0x07, 0x04, 0x4C, 0x00}, //nr. 5
                                  {0x06, 0x02, 0x20, 0x03}, //nr. 7
                                  {0x07, 0x05, 0x4C, 0x00}, //nr. 8
                                  {0x07, 0x05, 0x20, 0x03}}; //nr. 9
//=====
```

Cada número é escrito da seguinte forma: desloca-se o cursor de escrita para o endereço do *display* correspondente a primeira linha; escreve-se dois caracteres; desloca-se novamente o cursor, desta vez para a segunda linha, embaixo do primeiro caractere escrito; então, escreve-se os últimos dois caracteres. Na fig. 5.28, é apresentado o exemplo de um fluxograma simplificado para a escrita do número 5 com começo no endereço de caractere 0x80 (lado superior esquerdo do LCD, ver a fig. 5.21), considerando-se que os caracteres estão organizados conforme a tabela acima. Na sequência, apresenta-se o programa.

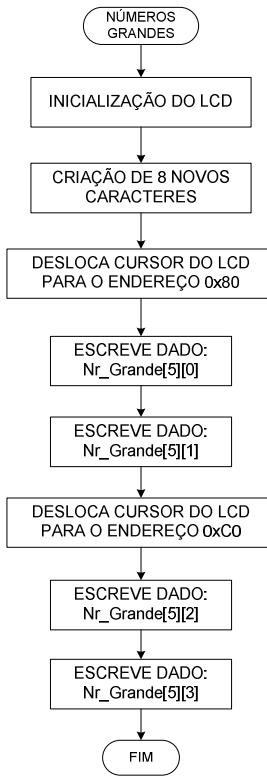


Fig. 5.28 – Fluxograma simplificado para apresentar o número grande 5 em um LCD 16 × 2.

LCD_4bits_big_number.c

```

//=====
//          CRIANDO NÚMEROS GRANDES PARA O LCD 16x2
//=====
#include "def_principais.h"      //inclusão do arquivo com as principais definições
#include "LCD.h"
//informações para criar novos caracteres, armazenadas na memória flash
const unsigned char novos_caract[] PROGMEM={0b00000001, 0b00000001, 0b00000001,
                                             0b00000001, 0b00000001, 0b00000001, 0b00011111, //0
                                             0b00001111, 0b00010000, 0b00010000, 0b00010000,
                                             0b00001000, 0b000010000, 0b00010000, //1
                                             0b00001111, 0b00000001, 0b00000001, 0b00000001,
                                             0b00000001, 0b00000001, 0b00000001, //2
                                             0b00000001, 0b00000001, 0b00000001, 0b00000001,
                                             0b00000001, 0b00000001, 0b00000001, //3
                                             0b00001111, 0b00000000, 0b00000000, 0b00000000,
                                             0b00000000, 0b00000000, 0b00011111, //4
                                             0b00001111, 0b00000001, 0b00000001, 0b00000001,
                                             0b00000001, 0b00000001, 0b00011111, //5

```

```

    0b00011111, 0b00000000, 0b00000000, 0b00000000,
    0b00000000, 0b00000000, 0b00000000, //6
    0b00011111, 0b00010000, 0b00010000, 0b00010000,
    0b00010000, 0b00010000, 0b00011111}//7

const unsigned char nr_grande[10][4] PROGMEM= {{0x01, 0x02, 0x4C, 0x00}, //nr. 0
                                                {0x20, 0x7C, 0x20, 0x7C}, //nr. 1
                                                {0x04, 0x05, 0x4C, 0x5F}, //nr. 2
                                                {0x06, 0x05, 0x5F, 0x00}, //nr. 3
                                                {0x4C, 0x00, 0x20, 0x03}, //nr. 4
                                                {0x07, 0x04, 0x5F, 0x00}, //nr. 5
                                                {0x07, 0x04, 0x4C, 0x00}, //nr. 5
                                                {0x06, 0x02, 0x20, 0x03}, //nr. 7
                                                {0x07, 0x05, 0x4C, 0x00}, //nr. 8
                                                {0x07, 0x05, 0x20, 0x03}};//nr. 9

//-----
void cria_novos_caract()//criação dos 8 novos caracteres
{
    unsigned char i, k, j=0, n=0x40;

    for(i=0;i<8;i++)
    {
        cmd_LCD(n,0);           //endereço base para gravar novo segmento
        for(k=0;k<7;k++)
            cmd_LCD(pgm_read_byte(&novos_caract[k+j]),1);
        cmd_LCD(0x00,1);/*apaga ultima posição do end. da CGRAM para evitar algum
                           dado espúrio*/
        j += 7;
        n += 8;
    }
}
//-----
void escreve_BIG(unsigned char end, unsigned char nr) //escreve um número grandes no LCD
{
    cmd_LCD(end,0);           //endereço de início de escrita (1a linha)
    cmd_LCD(pgm_read_byte(&nr_grande[nr][0]),1);
    cmd_LCD(pgm_read_byte(&nr_grande[nr][1]),1);
    cmd_LCD(end+64,0);       //desloca para a 2a linha
    cmd_LCD(pgm_read_byte(&nr_grande[nr][2]),1);
    cmd_LCD(pgm_read_byte(&nr_grande[nr][3]),1);
}
//-----
int main()
{
    DDRD = 0xFF;             //PORTD como saída
    DDRB = 0xFF;             //PORTB como saída
    inic_LCD_4bits();        //inicializa o LCD
    cria_novos_caract();     //cria os 8 novos caracteres

    //escreve os números 0, 1, 2, 3, 4 e 5
    escreve_BIG(0x80,0);
    escreve_BIG(0x82,1);
    escreve_BIG(0x85,2);
    escreve_BIG(0x88,3);
    escreve_BIG(0x8B,4);
    escreve_BIG(0x8E,5);

    for(;); //laço infinito
}
//=====

```

Na fig. 5.29, é apresentado o resultado para o programa supracitado. O detalhe está em escolher a posição correta para a escrita do número grande.



Fig. 5.29 – Números grandes num LCD 16 × 2.

Exercícios:

5.18 – Desenvolva um programa para realizar uma animação sequencial na linha superior e inferior de um LCD 16×2. Escreva um caractere por vez.

5.19 – Crie um caça-níquel eletrônico empregando 3 caracteres diferentes apresentados em 3 posições do LCD. Utilize um botão no pino PB2 do ATmega para o sorteio.

5.20 – Crie oito caracteres novos para o LCD 16×2. Comece a criar seu próprio conjunto de funções.

5.21 – Usando as duas linhas do LCD, crie um cronômetro com passo de 1 s. Utilize os números grandes (4 caracteres por dígito) e o pino PB2 do ATmega para o início/parada. Essa ideia é exemplificada na fig. 5.30.

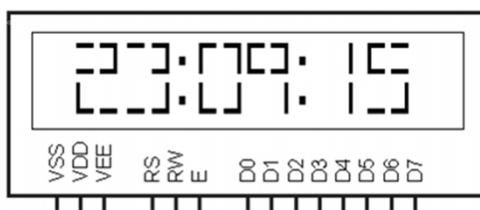


Fig. 5.30 – Números grandes para um cronômetro.

5.6 ROTINAS DE DECODIFICAÇÃO PARA USO EM DISPLAYS

Na programação em C, as variáveis podem ser de 8, 16 ou mais bits. Para apresentar em um *display* o valor de alguma dessas variáveis, é necessário decodificar o número representado pela variável em seus dígitos individuais. Por exemplo, supondo uma variável de 16 bits com o valor de 14569, como apresentar os dígitos 1, 4, 5, 6 e 9 em um LCD 16×2 ou em um conjunto de 5 *displays* de 7 segmentos?

A solução para o problema é dividir o número por 10, guardar o resto, pegar o número inteiro resultante e seguir com o processo até que a divisão resulte zero (ver os exemplos no capítulo 4). Os restos da divisão são os dígitos individuais do número (base decimal). A sub-rotina abaixo exemplifica o processo e serve para converter um número de 16 bits, sem sinal, nos seus cinco digitais individuais (valor máximo de 65.535).

```
-----  
#define tam_vetor 5 //número de dígitos individuais para a conversão por ident_num()  
#define conv_ascii 48 //48 se ident_num() deve retornar um número no formato ASCII (0 - normal)  
  
unsigned char digitos[tam_vetor];//declaração da variável para armazenagem dos dígitos  
-----  
void ident_num(unsigned int valor, unsigned char *disp)  
{  
    unsigned char n;  
  
    for(n=0; n<tam_vetor; n++)  
        disp[n] = 0 + conv_ascii;      //limpa vetor para armazenagem dos dígitos  
    do  
    {  
        *disp = (valor%10) + conv_ascii; //pega o resto da divisão por 10  
        valor /=10;                  //pega o inteiro da divisão por 10  
        disp++;  
    }  
    while (valor!=0);  
}  
//-----
```

A função recebe dois parâmetros: um deles é o **valor** a ser convertido, limitado ao tamanho da variável declarada, no caso acima, 16 bits (*unsigned int*); o outro parâmetro é o ponteiro para o vetor (***disp**) que conterá os valores individuais dos dígitos da conversão. No inicio da função, o vetor é zerado para garantir que valores anteriormente

convertidos não prejudiquem a conversão atual. Esse vetor deve ser declarado no corpo do programa onde será utilizado e seu tamanho é determinado pelo máximo valor que poderá conter.

O LCD com o controlador HD44780 entende somente caracteres no formato ASCII. Para ser impresso, o dado deve estar neste formato. Assim, para a conversão adequada, um número de base decimal necessita ser somado a 48 (0x30). Por exemplo, para apresentar o número 5 em uma posição do LCD, deve ser enviado o número 53 (conforme tab. B.4 do apêndice B). Em um *display* de 7 segmentos, é necessário utilizar a tabela de decodificação (tab. 5.2), como explicado na seção 5.4.

Abaixo, é apresentado um programa (baseado na seção 5.5.2) para a impressão de números de 0 a 100 em um LCD 16 × 2 empregando a função **ident_num(...)**.

LCD_4bits_ident_num.c

```
===== //  
//      ACIONANDO UM DISPLAY DE CRISTAL LIQUIDO DE 16x2          //  
//      Uso da função ident_num(...)                                //  
===== //  
#include "def_principais.h"    //inclusão do arquivo com as principais definições  
#include "LCD.h"  
//-----  
int main()  
{  
    unsigned char digitos[tam_vetor];//declaração da variável para armazenagem dos dígitos  
    unsigned char cont;  
    DDRD = 0xFF;                      //PORTD como saída  
    DDRB = 0xFF;  
    inic_LCD_4bits();                  //inicializa o LCD  
    while(1)  
    {  
        for(cont=0; cont<101; cont++)  
        {  
            ident_num(cont,digitos);  
            cmd_LCD(0x8D,0);//desloca o cursor para que os 3 dígitos fiquem a direita do LCD  
            cmd_LCD(digitos[2],1);  
            cmd_LCD(digitos[1],1);  
            cmd_LCD(digitos[0],1);  
            _delay_ms(200);           //tempo para troca de valor  
        }  
    }  
}=====
```

Dada a importância da estruturação na programação a função **ident_num(...)** foi agregada ao arquivo de funções LCD.c.

Outra conversão importante é transformar um número decimal em seus dígitos hexadecimais. Como os números são binários por natureza, basta isolar seus *nibbles*, onde cada conjunto de 4 bits é o próprio número hexadecimal, como exemplificado no código abaixo. Se o número a ser convertido tiver sinal ou não for inteiro, é preciso considerar esses fatores para realizar a conversão.

```
-----  
// Decodificando um número decimal em seus dígitos hexadecimais  
-----  
unsigned char num = 113;      //0x71  
unsigned char digit0, digit0;  
...  
digit0 = num & 0b00001111; /*mascara os 4 bits mais  
significativos, resultando no  
primeiro dígito (digit0 = 1)*/  
digit0 = num >> 4;          /*desloca o valor do número 4 bits  
para a direita,  
resultando no segundo dígito (4 mais  
significativos), digit0 = 7 */  
...  
-----
```

Exercícios:

5.22 – Crie um programa que conte até 10.000 apresentando a contagem decimal em um LCD 16×2 . Utilize um tempo de 100 ms para o incremento dos números, os dígitos devem ficar à direita do LCD e o número zero na frente do dígito mais significativo não deve ser apresentado.

5.23 – Repita o programa acima, só que desta vez, para apresentar um número hexadecimal em uma contagem até 0x3FF.

5.24 – Conte o número de vezes que um botão foi pressionado e apresente o valor em um LCD 16×2 . O botão pode ser conectado ao pino PB2 do ATmega e o circuito do LCD pode ser o mesmo da fig. 5.23a.

6. INTERRUPÇÕES

Neste capítulo, apresentam-se os conceitos relativos às interrupções, detalhando-se as interrupções externas provenientes de estímulos externos ao microcontrolador. As demais interrupções existentes no ATmega são tratadas nos capítulos subsequentes.

Por definição, interrupção é um processo pelo qual um dispositivo externo ou interno pode interromper a execução de uma determinada tarefa do microcontrolador e solicitar a execução de outra. Elas permitem que um programa responda a determinados eventos no momento em que eles ocorrem.

As interrupções são muito importantes no trabalho com microcontroladores porque permitem simplificar o código em determinadas situações, tornando mais eficiente o desempenho do processamento. Um exemplo clássico é o teclado de um computador: o processador não fica monitorando se alguma tecla foi pressionada; quando isso ocorre, o teclado gera um pedido de interrupção e a CPU interrompe o que está fazendo para executar a tarefa necessária. Assim, não há desperdício de tempo e de processamento.

6.1 INTERRUPÇÕES NO ATMEGA328

As interrupções no AVR são ‘vetoradas’, significando que cada interrupção tem um endereço de atendimento fixo na memória de programa. Se cada interrupção não possuísse esse endereço fixo, um único endereço seria destinado para todas as interrupções. Desta forma, para executar a tarefa de uma determinada interrupção, no caso de haver mais de uma habilitada, o programa teria que testar qual foi a interrupção ocorrida. Havendo um endereço fixo por interrupção, o programa se torna mais eficiente.

Todas as interrupções no AVR também são ‘mascaráveis’, ou seja, elas podem ser individualmente habilitadas ou desabilitadas agindo-se sobre um bit específico. O sistema de interrupção do AVR possui, ainda, um bit que pode desabilitar ou habilitar todas as interrupções de uma só vez, o bit I do registrador de status (SREG). Na tab. 6.1, apresenta-se o endereço de cada interrupção na memória de programa do ATmega328, existindo um total de 26 diferentes tipos de interrupções. A ordem dos endereços determina o nível de prioridade das interrupções, quanto menor o endereço do vetor de interrupção, maior será a sua prioridade. Por exemplo, a interrupção INTO tem prioridade sobre a INT1.

Tab. 6.1 – Interrupções do ATmega328 e seus endereços na memória de programa.

Vetor	End.	Fonte	Definição da Interrupção	Prioridade
1	0x00	RESET	Pino externo, Power-on Reset, Brown-out Reset e Watchdog Reset	
2	0x01	INT0	interrupção externa 0	
3	0x02	INT1	interrupção externa 1	
4	0x03	PCINT0	interrupção 0 por mudança de pino	
5	0x04	PCINT1	interrupção 1 por mudança de pino	
6	0x05	PCINT2	interrupção 2 por mudança de pino	
7	0x06	WDT	estouro do temporizador Watchdog	
8	0x07	TIMER2 COMPA	igualdade de comparação A do TC2	
9	0x08	TIMER2 COMPB	igualdade de comparação B do TC2	
10	0x09	TIMER2 OVF	estouro do TC2	
11	0x0A	TIMER1 CAPT	evento de captura do TC1	
12	0x0B	TIMER1 COMPA	igualdade de comparação A do TC1	
13	0x0C	TIMER1 COMPB	igualdade de comparação B do TC1	
14	0x0D	TIMER1 OVF	estouro do TC1	
15	0x0E	TIMER0 COMPA	igualdade de comparação A do TC0	
16	0x0F	TIMER0 COMPB	igualdade de comparação B do TC0	
17	0x10	TIMER0 OVF	estouro do TC0	
18	0x11	SPI, STC	transferência serial completa - SPI	
19	0x12	USART, RX	USART, recepção completa	
20	0x13	USART, UDRE	USART, limpeza do registrador de dados	
21	0x14	USART, TX	USART, transmissão completa	
22	0x15	ADC	conversão do ADC completa	
23	0x16	EE_RDY	EEPROM pronta	
24	0x17	ANA_COMP	comparador analógico	
25	0x18	TWI	interface serial TWI – I2C	
26	0x19	SPM_RDY	armazenagem na memória de programa pronta	

Quando o microcontrolador é inicializado, o contador de programa começa com o valor zero (endereço de *reset*). Se for empregado um programa de *boot loader*, o contador de programa é inicializado para o acesso à área da memória dedicada a ele. Então, somente depois do *boot loader* ser executado, inicia-se a execução do programa principal.

Toda vez que uma interrupção ocorre, a execução do programa é interrompida: a CPU completa a instrução em andamento, carrega na pilha o endereço da próxima instrução que seria executada (endereço de retorno) e desvia para a posição de memória correspondente à interrupção. O código escrito no endereço da interrupção é executado até o programa encontrar o código RETI (*Return from Interruption*). Então, é carregado no PC o endereço de retorno armazenado na pilha e o programa volta a trabalhar a partir do ponto que parou antes da ocorrência da interrupção.

Ao atender uma interrupção, o microcontrolador desabilita todas as outras interrupções que possam estar habilitadas, zerando o bit I do SREG (chave geral das interrupções). Ao retornar da interrupção (encontrar a instrução RETI), ele coloca novamente o bit I em 1, permitindo que outras interrupções sejam atendidas. Se uma ou mais interrupções ocorrerem neste período, os seus bits de sinalização serão colocados em 1 e o microcontrolador as tratará de acordo com a ordem de prioridade de cada uma. O detalhe é que o AVR executará sempre uma instrução do programa principal antes de atender qualquer interrupção em espera.

O AVR possui dois tipos de interrupção. O primeiro ativa um bit de sinalização (*flag*) indicando que a interrupção ocorreu, o qual é mantido em 1 até que a interrupção seja atendida, sendo zerado automaticamente pelo hardware. Alternativamente, o *flag* pode ser limpo pela escrita de 1 no bit correspondente. O bit de sinalização permite que várias interrupções fiquem ativas enquanto uma está sendo atendida e sejam processadas por ordem de prioridade. O segundo tipo de interrupção é disparada quando o evento que a gera está presente, não existem bits de sinalização e a interrupção só é atendida se sua condição existir quando a chave geral das

interrupções estiver ativa. Nesse caso, não há fila de espera, pois não há sinalização da ocorrência da interrupção. Este é o caso das interrupção externas por nível, por exemplo.

Quando uma interrupção é executada o registrador de estado (SREG) não é automaticamente preservado nem restaurado ao término de sua execução. Assim, **em assembly**, o registrador SREG e outros empregados no programa principal, e também em sub-rotinas (interrupção ou não), devem ter seus conteúdos salvos antes da execução da sub-rotina e restaurados ao término dessa. Caso contrário, o programa pode apresentar respostas imprevisíveis, visto que os valores dos registradores podem ter sido alterados fora da sequência normal do programa. Os comandos PUSH e POP são perfeitos para salvar e restaurar valores de registradores. Em C, **o salvamento dos registradores é feito automaticamente pelo compilador de forma transparente ao programador.**

Como as interrupções possuem endereços próximos, cada interrupção deve apresentar um comando de desvio para outra posição da memória (RJMP desvio) onde o código para o seu tratamento possa ser escrito. Caso contrário, quando ocorrer a interrupção o código executado não corresponderá ao desejado. Só existe a necessidade do desvio se a interrupção for utilizada.

Um exemplo de programação *assembly* com os endereços das interrupções é dada a seguir.

Endereço	Código	Comentários
	.ORG 0x00	;diretiva do assembly para gravar o código abaixo no end. 0x00
0x00	RJMP INICIO	;desvia para o início do programa
0x01	RJMP EXT_INT0	;interrup. externa 0
0x02	RJMP EXT_INT1	;interrup. externa 1
0x03	RJMP PCINT0	;interrup. 0 por mudança de pino
0x04	RJMP PCINT1	;interrup. 1 por mudança de pino
0x05	RJMP PCINT2	;interrup. 2 por mudança de pino
0x06	RJMP WDT	;estouro do temporizador Watchdog
0x07	RJMP TIM2_COMPA	;igualdade de comparação A do TC2
0x08	RJMP TIM2_COMPB	;igualdade de comparação B do TC2
0x09	RJMP TIM2_OVF	;estouro do TC2
0x0A	RJMP TIM1_CAPT	;evento de captura do TC1
0x0B	RJMP TIM1_COMPA	;igualdade de comparação A do TC1

```

0x0C      RJMP TIM1_COMPB      ;igualdade de comparação B do TC1
0x0D      RJMP TIM1_OVF       ;estouro do TC1
0x0E      RJMP TIM0_COMPA     ;igualdade de comparação A do TC0
0x0F      RJMP TIM0_COMPB     ;igualdade de comparação B do TC0
0x10      RJMP TIM0_OVF       ;estouro do TC0
0x11      RJMP SPI_STC        ;transferência serial completa - SPI
0x12      RJMP USART_RX        ;USART, recepção completa
0x13      RJMP USART_UDRE      ;USART, limpeza do registr. de dados
0x14      RJMP USART_TX        ;USART, transmissão completa
0x15      RJMP ADC             ;conversão do ADC completa
0x16      RJMP EE_RDY          ;EEPROM pronta
0x17      RJMP ANALOG_COMP    ;comparador analógico
0x18      RJMP TWI              ;interface serial TWI
0x19      RJMP SPM_RDY          ;memór. de armaz. de programa pronta
...
INICIO:    ... ;código para o início, programa principal
EXT_INT0:   ... ;código para a interrup. externa 0
...

```

No compilador AVR-GCC, a tabela de vetores de interrupção é pré-definida para apontar para rotinas de interrupção com nomes pré-determinados. Usando o nome apropriado, a rotina será chamada quando ocorrer a interrupção correspondente. O AVR-GCC emprega os nomes abaixo para os tratadores de interrupção. As interrupções são escritas como funções e podem estar antes ou depois do programa principal (**main()**).

Código (função)

```

int main()           //aqui vai o programa principal
ISR(INT0_vect)      //interrupção externa 0
ISR(INT1_vect)      //interrupção externa 1
ISR(PCINT0_vect)    //interrupção 0 por mudança de pino
ISR(PCINT1_vect)    //interrupção 1 por mudança de pino
ISR(PCINT2_vect)    //interrupção 2 por mudança de pino
ISR(WDT_vect)        //estouro do temporizador Watchdog
ISR(TIMER2_COMPA_vect) //igualdade de comparação A do TC2
ISR(TIMER2_COMPB_vect) //igualdade de comparação B do TC2
ISR(TIMER2_OVF_vect)  //estouro do TC2}
ISR(TIMER1_CAPT_vect) //evento de captura do TC1
ISR(TIMER1_COMPA_vect) //igualdade de comparação A do TC1
ISR(TIMER1_COMPB_vect) //igualdade de comparação B do TC1
ISR(TIMER1_OVF_vect)  //estouro do TC1
ISR(TIMER0_COMPA_vect) //igualdade de comparação A do TC0
ISR(TIMER0_COMPB_vect) //igualdade de comparação B do TC0
ISR(TIMER0_OVF_vect)  //estouro do TC0
ISR(SPI_STC_vect)    //transferência serial completa - SPI
ISR(USART_RX_vect)   //USART, recepção completa
ISR(USART_UDRE_vect) //USART, limpeza do registrador de dados
ISR(USART_TX_vect)   //USART, transmissão completa
ISR(ADC_vect)         //conversão do ADC completa
ISR(EE_READY_vect)   //EEPROM pronta
ISR(ANALOG_COMP_vect) //comparador analógico
ISR(TWI_vect)         //interface serial TWI
ISR(SPM_READY_vect)  //armazenagem na memória de programa pronta
}

```

Existem registradores específicos e geral de controle das interrupções. Cada interrupção é habilitada por um bit específico no seu registrador de controle e a interrupção é efetivamente ativada quando o bit I do registrador SREG for colocado em 1. Na fig. 6.1, são ilustradas as chaves de habilitação das interrupções. A chave geral é utilizada para ligar ou desligar todas as interrupções ativas de uma única vez. No AVR-GCC, a função `sei()` liga a chave geral das interrupções e a função `cli()` a desliga. Os bits individuais das interrupções serão vistos em momentos oportunos.

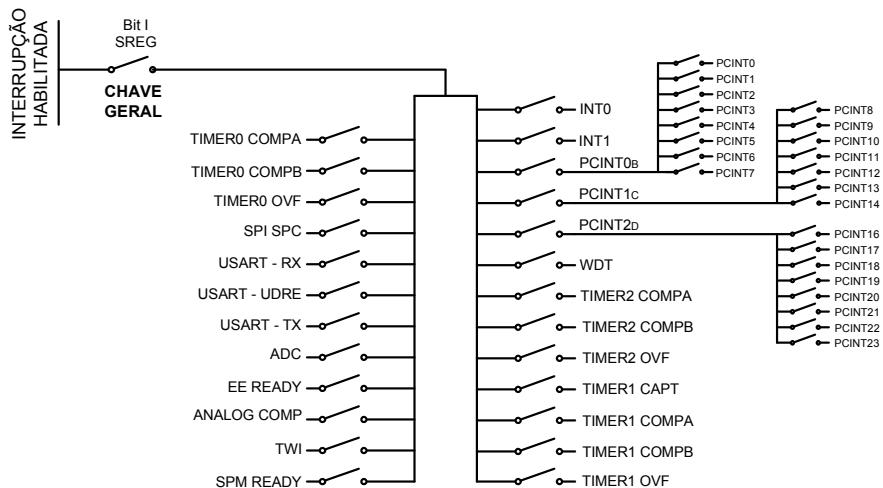


Fig. 6.1 – Chaves de habilitação das interrupções.

6.2 INTERRUPÇÕES EXTERNAS

As interrupções externas são empregadas para avisar o microcontrolador que algum evento externo a ele ocorreu, o qual pode ser um botão pressionado ou um sinal de outro sistema digital. Muitas vezes, elas são empregadas para retirar a CPU de um estado de economia de energia, ou seja, para ‘despertar’ o microcontrolador.

Todos os pinos de I/O do ATmega328 podem gerar interrupções externas por mudança de estado lógico no pino; o nome delas nos pinos são PCINT0 até PCINT23. Entretanto, existem dois pinos, INT0 e INT1, que podem gerar interrupções na borda de subida, descida ou na manutenção do nível do estado lógico no pino. Se habilitada, a interrupção pode ocorrer mesmo que o pino esteja configurado como saída, permitindo gerar interrupção por software ao se escrever no pino.

Cada uma das interrupções para os pinos INT0 e INT1 possuem um endereço fixo na memória de programa; já as interrupções PCINTx possuem somente 3 endereços, um para cada PORT. Caso mais de um pino no PORT esteja habilitado para gerar a interrupção, é necessário, via programação, testar qual pino gerou a interrupção.

O registrador de controle EICRA (*Extern Interrupt Control Register A*) contém os bits responsáveis pela configuração das interrupções externas INT0 e INT1, conforme tab. 6.2.

Bit	7	6	5	4	3	2	1	0
EICRA	-	-	-	-	ISC11	ISC10	ISC01	ISC00
Lê/Escrve	L	L	L	L	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Tab. 6.2 – Bits de configuração da forma das interrupções nos pinos INT1 e INTO.

ISC11	ISC10	Descrição
0	0	Um nível baixo em INT1 gera um pedido de interrupção.
0	1	Qualquer mudança lógica em INT1 gera um pedido de interrupção.
1	0	Uma borda de decida em INT1 gera um pedido de interrupção.
1	1	Uma borda de subida em INT1 gera um pedido de interrupção.
ISC01	ISC00	Descrição
0	0	Um nível baixo em INTO gera um pedido de interrupção.
0	1	Qualquer mudança lógica em INTO gera um pedido de interrupção.
1	0	Uma borda de decida em INTO gera um pedido de interrupção.
1	1	Uma borda de subida em INTO gera um pedido de interrupção.

As interrupções são habilitadas nos bits INT1 e INTO do registrador EIMSK (*External Interrupt Mask Register*) se o bit I do registrador SREG (*Status Register*) também estiver habilitado.

Bit	7	6	5	4	3	2	1	0
EIMSK	-	-	-	-	-	-	INT1	INT0
Lê/Escrive	L	L	L	L	L	L	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

O registrador EIFR (*External Interrupt Flag Register*) contém os dois bits sinalizadores que indicam se alguma interrupção externa ocorreu. São limpos automaticamente pela CPU após a interrupção ser atendida. Alternativamente, esses bits podem ser limpos pela escrita de 1 lógico.

Bit	7	6	5	4	3	2	1	0
EIFR	-	-	-	-	-	-	INTF1	INTF0
Lê/Escrive	L	L	L	L	L	L	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

O registrador PCICR (*Pin Change Interrupt Control Register*) é responsável pela habilitação de todas as interrupções externas nos pinos de I/O, do PCINT0 até o PCINT23, as quais são divididas entre os três PORTs do microcontrolador:

Bit	7	6	5	4	3	2	1	0
PCICR	-	-	-	-	-	PCIE2	PCIE1	PCIE0
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

onde PCIE0 habilita a interrupção por qualquer mudança nos pinos do PORTB (PCINT0:7), PCIE1 nos pinos do PORTC (PCINT8:14) e PCIE2 nos pinos do PORTD (PCINT16:23).

Quando ocorrer um interrupção em algum dos pinos, o bit de sinalização do PORT no registrador PCIFR é ativo: PCIF0 para o PORTB, PCIF1 para o PORTC e PCIF2 para o PORTD.

Bit	7	6	5	4	3	2	1	0
PCIFR	-	-	-	-	-	PCIF2	PCIF1	PCIF0
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Com pode ser observado na fig. 6.1, existem várias chaves ligadas às chaves de interrupção dos PORTs, PCINT0_B, PCINT1_C e PCINT2_D, as quais indicam que cada pino de I/O pode ser individualmente habilitado para gerar uma interrupção externa. É importante observar que a Atmel usa os mesmos nomes PCINT0, PCINT1 e PCINT2 para nomear as interrupções dos PORTs e, ainda, para três pinos, gerando confusão. Desta forma, os pinos PCINT0, 1 e 2 estão ligados ao PORTB e pertencem somente à interrupção PCINT0. A habilitação individual dos pinos é feita nos registradores PCMSK0:2.

Bit	7	6	5	4	3	2	1	0
PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
Lê/Escrive	L/E							
Valor Inicial	0	0	0	0	0	0	0	0

Para ilustrar a operação das duas interrupções externas, INTO e INT1, configuradas, respectivamente, para gerar interrupção por nível e por transição, é apresentado na fig. 6.2 um circuito exemplo. Um botão irá trocar o estado do LED (se ligado, desliga e vice-versa), o outro botão mantido pressionado piscará o LED. O código exemplo é apresentado em seguida.

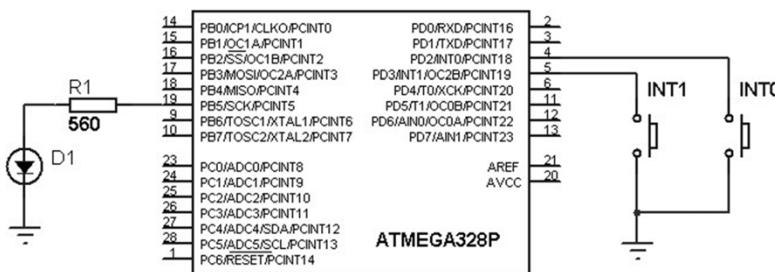


Fig. 6.2 – Circuito paraemploi das interrupções externas INT0 e INT1.

INTO 1.c

```
//===================================================================== //  
// HABILITANDO AS INTERRUPÇÕES INT0 e INT1 POR TRANSIÇÃO E NÍVEL, RESPECTIVAMENTE //  
//===================================================================== //  
  
#define F_CPU 16000000UL  
#include <avr/io.h>  
#include <util/delay.h>  
#include <avr/interrupt.h>  
  
//Definições de macros - empregadas para o trabalho com bits  
#define set_bit(Y,bit_X) (Y|=(1<<bit_X)) //ativa o bit x da variável  
#define clr_bit(Y,bit_X) (Y&=~(1<<bit_X)) //limpa o bit x da variável  
#define tst_bit(Y,bit_X) (Y&(1<<bit_X)) //testa o bit x da variável  
#define cpl_bit(Y,bit_X) (Y^=(1<<bit_X)) //troca o estado do bit x  
  
#define LED PB5 //LED está no pino PB5
```

```

ISR(INT0_vect);
ISR(INT1_vect);

//-----
int main()
{
    DDRD = 0x00;           //PORTD entrada
    PORTD = 0xFF;          //pull-ups habilitados
    DDRB = 0b00100000;    //somente pino do LED como saída
    PORTB = 0b11011111;   //desliga LED e habilita pull-ups
    UCSR0B = 0x00;         /*necessário desabilitar RX e TX para trabalho com os pinos
                           do PORTD no Arduino*/
    EICRA = 1<<ISC01;//interrupções externas: INT0 na borda de descida, INT1 no nível zero.
    EIMSK = (1<<INT1) | (1<<INT0);//habilita as duas interrupções
    sei();                 //habilita interrupções globais, ativando o bit I do SREG

    while(1){}
}

//-----
ISR(INT0_vect) //interrupção externa 0, quando o botão é pressionado o LED troca de estado
{
    cpl_bit(PORTB,LED);
}

//-----
ISR(INT1_vect) //interrupção externa 1, mantendo o botão pressionado o LED pisca
{
    cpl_bit(PORTB,LED);
    _delay_ms(200);      //tempo para piscar o LED
}
//=====

```

No programa acima, quando o botão ligado a INT0 é pressionado, o LED troca de estado (liga ou desliga). A interrupção foi habilitada para ocorrer somente na transição de 1 para 0 e, como não existe tratamento para o ruído gerado no acionamento do botão, a resposta pode não ser a desejada. Esse exemplo mostra que o uso de um botão com interrupção pode não ser adequado para algumas aplicações. Por sua vez, o botão ligado a interrupção INT1 foi habilitado para gerar uma interrupção por nível lógico 0, ou seja, enquanto o botão estiver pressionado a interrupção irá ocorrer e a rotina de tratamento da interrupção continuará sendo executada. Dessa forma, como existe um atraso de 200 ms dentro da interrupção, mantendo-se o botão pressionado, o LED irá piscar e não existe prejuízo devido ao ruído do botão.

Ao se utilizar rotinas de interrupção, deve-se evitar códigos extensos, pois o programa principal (**int main()**) fica suspenso até o término da execução da rotina. O programador deve avaliar o impacto do tempo gasto

dentro da rotina de interrupção em relação ao restante do programa. Outro ponto importante a se observar é evitar, sempre que possível, chamar outras funções dentro da interrupção, pois isso degrada o desempenho do programa e pode gerar problemas difíceis de detectar.

O circuito da fig. 6.3 e o seu respectivo programa de teste são empregados para exemplificar o uso das interrupções externas por mudança em qualquer pino de I/O. Quando um botão do PORTC é pressionado, o LED definido para o pino, liga ou desliga. Como as interrupções são por mudança de estado no pino, é gerado um pedido de interrupção sempre que se pressiona e se solta o botão. Existindo ruído no acionamento do botão, o resultado será imprevisível. Assim, foi colocado um atraso de 200 ms no final da rotina de interrupção para que o botão seja pressionado rapidamente, ativando a interrupção somente na borda de descida do sinal; caso contrário, o estado do LED seria aleatório. Como a interrupção externa PCINT1 só possui um endereço na memória de programa, é necessário testar qual foi o pino responsável pela interrupção.

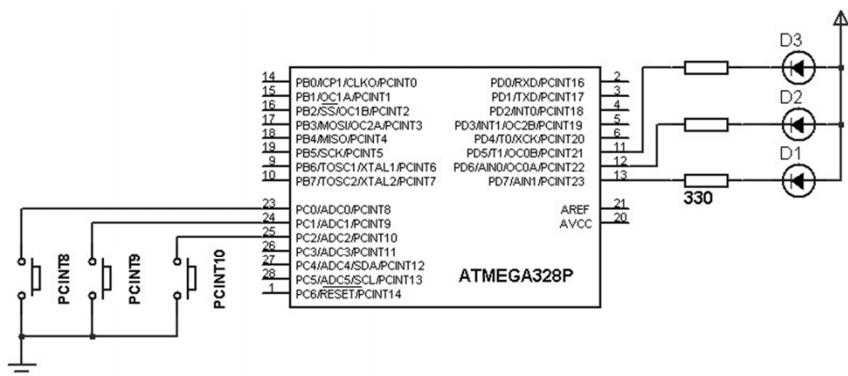


Fig. 6.3 – Circuito para teste das interrupções externas PCINT8:10.

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL      //frequência de trabalho
#include <avr/io.h>          //definições do componente especificado
#include <avr/interrupt.h>    //define algumas macros para as interrupções
#include <util/delay.h>       //biblioteca para o uso das rotinas de delay

//Definições de macros para o trabalho com bits
#define set_bit(y,bit) (y|=(1<<bit))//coloca em 1 o bit x da variável Y
#define clr_bit(y,bit) (y&=~(1<<bit))//coloca em 0 o bit x da variável Y
#define cpl_bit(y,bit) (y^=(1<<bit))//troca o estado lógico do bit x da variável Y
#define tst_bit(y,bit) (y&(1<<bit))//retorna 0 ou 1 conforme leitura do bit

#define LED0 PD5
#define LED1 PD6
#define LED2 PD7

#endif
```

PCINTx.c (programa principal)

```
//===================================================================== //
//   Cada vez que um botão é pressionado o LED correspondente troca de estado      //
//===================================================================== //
#include "def_principais.h"//inclui arquivo com as definições principais

ISR(PCINT1_vect);

int main()
{
    DDRD = 0x00;           //PORTC como entrada, 3 botões
    PORTC = 0xFF;          //habilita pull-ups
    DDRD = 0b11100000;    //pinos PD5:7 do PORTC como saída (leds)
    PORTD = 0xFF;          //apaga leds e habilita pull-ups dos pinos não utilizados

    PCICR = 1<<PCIE1;//habilita interrupção por qualquer mudança de sinal no PORTC
    PCMSK1 = (1<<PCINT10)|(1<<PCINT9)|(1<<PCINT8);/*habilita os pinos PCINT8:10 para
                                                 gerar interrupção*/
    sei();                //habilita as interrupções

    while(1){}
}

//-----
ISR(PCINT1_vect)
{
    //quando houver mais de um pino que possa gerar a interrupção é necessário testar qual foi
    if(!tst_bit(PINC,PC0))
        cpl_bit(PORTD,LED0);
    else if(!tst_bit(PINC,PC1))
        cpl_bit(PORTD,LED1);
    else if(!tst_bit(PINC,PC2))
        cpl_bit(PORTD,LED2);

    _delay_ms(200);
}
//=====================================================================
```

6.2.1 UMA INTERRUPÇÃO INTERROMPENDO OUTRA

O ATmega não suporta interrupções aninhadas, isto é, uma interrupção não pode interromper outra em andamento, mesmo que tenha maior prioridade. Ocorrendo uma ou mais interrupções enquanto uma está sendo tratada, é formado uma fila de espera que é atendida pela ordem de prioridade. Dessa forma, ao tratar uma interrupção, a CPU automaticamente desabilita todas as interrupções, voltando a ligá-las ao final da rotina de interrupção.

Para fazer o aninhamento de interrupções é necessário, via programação, habilitar a chave geral das interrupções, o bit I do registrador SREG, dentro da interrupção que se deseja interromper. Nesse caso, deve-se ter cuidado em preservar o registrador de *status* – SREG e analisar a prioridade das interrupções para que o programa não gere respostas inesperadas. A seguir, é apresentado um exemplo que ilustra esta ideia: uma interrupção de menor prioridade interrompendo uma de maior, INT1 versus INT0, ambas habilitadas para gerar a interrupção por nível zero. O arquivo **def_principais.h** é o mesmo apresentado anteriormente. Na fig. 6.4 pode-se ver o circuito de teste montado para o Arduino.

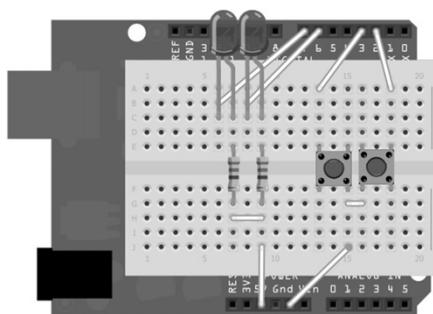


Fig. 6.4 – Circuito montado no Arduino para teste das interrupções INT0 e INT1.

INTO_1_aninhada.c

```
//=====
//  INTERRUPÇÃO INT1 INTERROMPENDO A INTO
//=====
#include "def_principais.h"

ISR(INT0_vect);
ISR(INT1_vect);

//-----
int main()
{
    DDRD = 0b11000000;//configurando os pinos de entrada e saída
    PORTD = 0b11111111;//desligando leds e habilitando pull-ups

    UCSR0B = 0x00; //desabilitando RX e TX para trabalho com os pinos do Arduino
    EICRA = 0x00; //interrupções externas INT0 e INT1 no nível zero.
    EIMSK = (1<<INT1)|(1<<INT0); //habilita as duas interrupções
    sei(); //habilita interrupções globais, ativando o bit I do SREG

    while(1)//pisca led numa velocidade muito grande (visualmente fica ligado)
        cpl_bit(PORTD,LED2);
}

//-----
ISR(INT0_vect) //interrupção externa 0, quando o botão é pressionado o LED pisca
{
    unsigned char sreg;

    sreg = SREG; //salva SREG porque a interrupção pode alterar o seu valor
    clr_bit(EIMSK,INT0);//desabilita INTO para que ele não chame a si mesmo
    sei(); //habilita a interrupção geral, agora INT1 pode interromper INTO

    cpl_bit(PORTD,LED1); //pisca led a cada 300 ms
    _delay_ms(300);

    set_bit(EIMSK,INT0); //habilita novamente a interrupção INTO

    SREG = sreg; //restaura o valor de SREG que pode ter sido alterado
}
//-----
ISR(INT1_vect) //interrupção externa 1, mantendo o botão pressionado o LED pisca
{
    cpl_bit(PORTD,LED1); //pisca led a cada 200 ms
    _delay_ms(200);
}
=====
```

Quando INTO está ativa, INT1 consegue interrompê-la, isso pode ser verificado pelo aumento na frequência que o LED1 pisca. Dentro da rotina de interrupção INTO alguns passos são necessários: **primeiro, salvar o conteúdo do SREG, pois o compilador não sabe que pode haver outra interrupção que altere o seu valor;** **segundo, desabilitar a própria interrupção INTO, pois ela é prioritária e estando habilitada por nível vai gerar um pedido de interrupção ainda dentro da sua própria rotina,**

entrando para a fila de prioridade de execução; terceiro, executar o que se deseja como ação para a interrupção; quarto, habilitar novamente a interrupção INT0 e; quinto, restaurar o valor original do SREG.

No código de atendimento de INT1 é necessário somente as instruções para a ação desejada, não é necessário nenhum cuidado especial com o SREG porque o compilador AVR-GCC automaticamente salva o seu conteúdo na entrada de uma interrupção e o retorna na saída.

Exercícios:

- 6.1** – Supondo que sejam habilitadas 3 interrupções: INT0, INT1 e PCINT0, e que INT0 e PCINT0 sempre estão ativas ao término da interrupção INT1, quando a interrupção PCINT0 será executada?
- 6.2** – Teste o exemplo da interrupção INT1 interrompendo a INT0 (seção 6.2.1). Por que se consegue ver que o LED2 do laço principal está piscando quando um dos botões é mantido pressionado? Por que a interrupção INT0 não consegue interromper a INT1?
Exclua o trecho de código dentro da interrupção INT0 que salva o SREG e o restaura. O que acontece?
- 6.3** – Faça um programa para testar a interrupção PCINT2, usando dois botões nos pinos PD6 e PD7, onde cada um deve ligar um LED nos pinos PD0 e PD1.
-

7. GRAVANDO A EEPROM

A memória EEPROM é um importante periférico disponível no ATmega. Neste capítulo, são apresentadas as características e formas de programação da EEPROM do ATmega328.

O ATmega possui 4 tipos diferentes de memórias (apresentadas no capítulo 2), é fundamental compreendê-las para evitar confusões. Assim:

1. O programa é salvo na memória *flash*, por isso ela é conhecida como memória de programa (32 kbytes no ATmega328).
2. Na memória RAM são armazenadas as variáveis temporárias, aquelas variáveis declaradas no programa e que perdem seu valor na desenergização do circuito (2 kbytes no ATmega328, é a SRAM).
3. Para o trabalho da CPU e de seus periféricos, é empregada a memória de dados ou de I/O. Nela estão os registradores de trabalho do microcontrolador. O programador pode ler ou alterar o valor dos seus bits, ela é o ‘painel’ de controle do microcontrolador.
4. Para salvar dados importantes que devem ser preservados após a desenergização do circuito e estarem disponíveis na reinicialização do sistema, é empregada a memória EEPROM.

A memória EEPROM é utilizada com dados que podem ser alterados e devem estar disponíveis para o programa, ou seja, não podem ser perdidos. A EEPROM é como um *pendrive* do microcontrolador. Pode ser utilizada, por exemplo, para a armazenagem de senhas, variáveis lidas pelo conversor AD, valor de tempo configurado para alarmes ou quaisquer valores que devam ser salvos. Em resumo, a EEPROM armazena dados para futura leitura pelo programa.

A EEPROM é uma memória de escrita lenta, o que deve ser levado em conta quando for utilizada. O tempo de escrita de um dado é de aproximadamente 3,3 ms, o que pode ser longo demais para certas

aplicações. Caso esse tempo seja crítico, pode-se empregar a interrupção da EEPROM para avisar quando uma escrita foi completada, liberando o programa dessa monitoração.

A EEPROM não deve ser utilizada para a escrita de dados estáticos que nunca serão alterados²⁸; para essa função deve-se utilizar a memória *flash*, a menos que haja limitações de espaço nessa memória.

O ATmega328 possui 1 kbyte de memória EEPROM, a qual é organizada em um espaço de memória separado, em que bytes individuais podem ser lidos e escritos. A EEPROM suporta pelo menos 100 mil ciclos de escrita e apagamento. A comunicação entre a EEPROM e a CPU é feita pelos registradores de endereço, de dados e de controle da EEPROM.

De acordo com os registradores e bits de trabalho da EEPROM, os seguintes passos são necessários para efetuar-se uma escrita (a ordem dos passos 2 e 3 não é importante). Esses passos são necessários para se evitar um eventual escrita accidental.

1. Esperar até que o bit EEPE do registrador EECR se torne zero.
2. Escrever o novo endereço da EEPROM no registrador EEAR (opcional).
3. Escrever o dado a ser gravado no registrador EEDR (opcional).
4. Escrever 1 lógico no bit EEMPE enquanto escreve 0 no bit EEPE do registrador EECR.
5. Dentro de quatro ciclos de *clock* após ativar EEMPE, escrever 1 lógico no bit EEPE.

Os exemplos de códigos a seguir assumem que nenhuma interrupção irá ocorrer durante a execução das funções. É importante não haver interrupções habilitadas durante a escrita para evitar eventuais erros.

²⁸ A EEPROM possui um custo de fabricação caro comparado as outras memórias do microcontrolador. É por isso que alguns microcontroladores de 32 bits conseguem competir em preços com os 8 bits ao não incorporar o processo de fabricação da EEPROM nos seus chips.

Código em Assembly:

EEPROM_escrita:

```
SBIC EECR,EEPE      ;espera completar um escrita prévia
RJMP EEPROM_escrita
OUT EEARH, R18       ;escreve o end. (R18:R17) no registr. de end.
OUT EEARL, R17
OUT EEDR,R16         ;escreve o dado (R16) no registrador de dado
SBI EECR,EEMPE      ;escreve um lógico em EEMPE
SBI EECR,EEPE        ;inicia a escrita colocando o bit EEPE em 1
RET
```

EEPROM_leitura:

```
SBIC EECR,EEPE      ;espera completar um escrita prévia
RJMP EEPROM_leitura
OUT EEARH, R18       ;escreve o end.(R18:R17) no registr. de end.
OUT EEARL, R17
SBI EECR,EERE        ;inicia leitura escrevendo em EERE
IN  R16,EEDR         ;lê dado do registrador de dados em R16
RET
```

Código em C:

```
void EEPROM_escrita(unsigned int uiEndereco, unsigned char ucDado)
{
    while(EECR & (1<<EEPE)); //espera completar um escrita prévia
    EEAR = uiEndereco;        //carrega o endereço para a escrita
    EEDR = ucDado;           //carrega o dado a ser escrito
    EECR |= (1<<EEMPE);     //escreve um lógico em EEMPE
    EECR |= (1<<EEPE);      //inicia a escrita ativando EEPE
}

unsigned char EEPROM_leitura(unsigned int uiEndereco)
{
    while(EECR & (1<<EEPE)); //espera completar um escrita prévia
    EEAR = uiEndereco;        //escreve o endereço de leitura
    EECR |= (1<<EERE);      //inicia a leitura ativando EERE
    return EEDR;              //retorna o valor lido do registrador de
                               //dados
}
```

Para a programação em C da EEPROM no AVR Studio, basta incluir no arquivo fonte a biblioteca `<avr/eeprom.h>` do AVR-GCC, a qual possui várias funções para o trabalho com a EEPROM. Uma importante questão na gravação de dados, é que muitas vezes eles devem ser definidos previamente à gravação da memória de programa (inicialização da EEPROM). Este é o caso, por exemplo, quando se deseja empregar dados

pré-definidos com alguma função. Então, a gravação dos dados para a EEPROM é feita no momento da gravação do microcontrolador, não consumindo memória de programa (*flash*). O trabalho do programador é definir quais dados serão gravados. A seguir, é apresentado um exemplo em C para a gravação e leitura de dados na EEPROM. Detalhes para a inicialização da EEPROM na gravação do microcontrolador são apresentados no capítulo 23.

EEPROM.c

```
//=====================================================================
//      GRAVANDO E LENDO A EEPROM                               //
//=====================================================================

#include <avr/io.h>
#include <avr/eeprom.h>
//------------------------------------------------------------------------------

/*initialização dos valores para a EEPROM começando no endereço 0x00 (default), não
consome nenhum byte da memória flash - deve ser empregado quando necessário.
Gera um arquivo *.eep que é utilizado no programa de gravação do microcontrolador*/

unsigned char EEMEM uc_valor = 0x33;
unsigned char EEMEM uc_vetor[4] = {0x33, 0x22, 0x11, 0x55};
unsigned char EEMEM uc_string[13] = {"Teste EEPROM\0"};
unsigned int EEMEM ui_valor = 0x5AB9;
//------------------------------------------------------------------------------

unsigned char RAM_byte;
unsigned char RAM_bytes[4];
unsigned char RAM_string[13];
unsigned int RAM_word;
//------------------------------------------------------------------------------

int main()
{
    unsigned char ucDado_escrita = 0x13;
    unsigned char ucDado_leitura;
    unsigned int uiEndereco = 0x3FF; /*endereço 0x3FF (1024º posição da memória)
                                         valores entre 0 e 1023*/
    _EEPPUT(uiEndereco,ucDado_escrita); //gravando 0x13 no endereço 0x3FF da EEPROM
    _EEGET(ucDado_leitura, uiEndereco);/*lendo o conteúdo do endereço 0x3FF para a
                                         variável ucDado_leitura*/
    //lendo valores da EEPROM para a RAM
    RAM_byte = eeprom_read_byte(&uc_valor);/*lendo a variável uc_valor para a
                                         variável RAM_byte*/
    RAM_word = eeprom_read_word(&ui_valor);/*lendo a variável ui_valor para a
                                         variável RAM_word*/
    eeprom_read_block(RAM_string,uc_string,13);/*lendo a variável uc_string com 13
                                         bytes para a variável RAM_string*/
    eeprom_read_block(RAM_bytes,uc_vetor,4);/*lendo a variável uc_vetor com 4 bytes
                                         para a variável RAM_bytes*/
    //escrevendo um dado na EEPROM
    eeprom_write_byte(&uc_valor,0xEE);/*escreve o valor 0xEE na variável uc_valor da
                                         EEPROM*/
    while(1){}/código principal
}

//=====================================================================
```

Com o uso da biblioteca **eprom.h**, a palavra chave EEMEM indica ao compilador que as variáveis têm que ser armazenadas na EEPROM. Assim, essas variáveis são salvas em um arquivo com extensão *.eep. Nesse caso, não é consumido nenhum byte da memória de programa. Quando for empregado o hardware específico de gravação do microcontrolador, esse arquivo deve ser gravado na EEPROM, inicializando-a com os seus valores. Para escrever um dado, basta utilizar a macro _EEPUT, passando para a função o endereço onde se quer escrever e o dado; para a leitura, empregase a macro _EGET, passando o nome da variável onde será salvo o dado lido e o endereço que se deseja ler. Essas duas macros foram definidas no AVR-GCC para manter a compatibilidade com o compilador IARC. O AVR-GCC oferece inúmeras funções para o gerenciamento de dados entre a EEPROM e a memória RAM, algumas exemplificadas no código anterior.

Exercícios:

7.1 – Faça um programa para gravar 0xAA em toda a memória EEPROM do ATmega328. Certifique-se que os dados foram corretamente gravados, lendo-os após a gravação. Pode ser empregado um *display* para informar sobre o sucesso da operação ou um LED. Depois disso, com base no manual do ATmega328, empregue a interrupção da EEPROM para repetir o processo.

7.2 – Supondo que exista um circuito integrado ligado ao ATmega328 e que a primeira vez que o sistema é energizado esse circuito necessite uma inicialização específica. O programa do microcontrolador deve interagir com o usuário e perguntar se a inicialização do circuito integrado deve ser feita; caso afirmativo, essa ação será feita somente uma vez e, na próxima vez que o microcontrolador for inicializado, o programa não fará mais nenhuma pergunta.

Faça um programa que utilize um byte da EEPROM para armazenar a informação de que a inicialização do circuito integrado foi feita. Empregue dois LEDs e um botão. A primeira vez, um dos LEDs acende, indicando que a inicialização necessita ser feita; quando o botão for pressionado, o segundo LED deve ser ligado. Ao se reenergizar o circuito, somente o segundo LED deve ficar ligado, indicando que a inicialização já foi feita e o processo de inicialização não é mais necessário.

8. TECLADO MATRICIAL

Neste capítulo, apresenta-se a técnica de varredura para um teclado matricial. Essa técnica é comum em sistemas microcontrolados, sendo empregada para maximizar o uso dos pinos de I/O do microcontrolador.

Uma forma muito comum de entrada de dados em um sistema microcontrolado é através de teclas (botões ou chaves tátteis). Quando o número delas é pequeno, cada uma pode ser associada a um pino de I/O do microcontrolador. Entretanto, quando o seu número é grande, não é conveniente utilizar muitos pinos de I/O. Um teclado convencional emprega 3×4 teclas (12) ou 4×4 teclas (16) (fig. 8.1). Ao invés de se empregar 12 ou 16 pinos para a leitura desses teclados, empregam-se 7 ou 8 pinos, respectivamente.

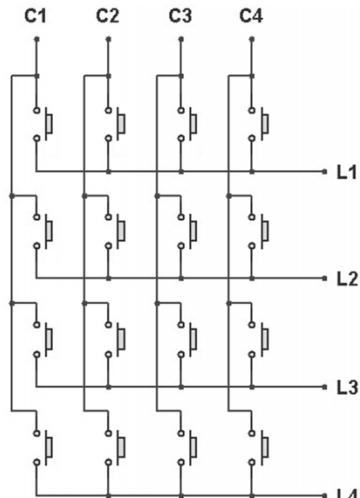
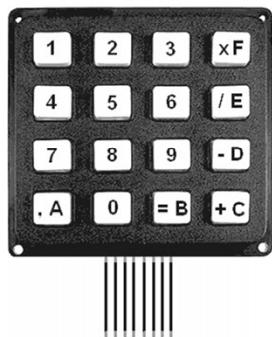


Fig. 8.1 – Teclado matricial hexadecimal: 4×4 .

Para a leitura de teclados com várias teclas, é necessário empregar o conceito de matriz, em que os botões são arranjados em colunas e linhas, conforme é mostrado na fig. 8.1; o teclado é lido utilizando-se uma varredura. O pressionar de uma tecla produzirá um curto-circuito entre uma coluna e uma linha e o sinal da coluna se refletirá na linha ou vice-versa. A ideia é trocar sucessivamente o nível lógico das colunas, a chamada varredura, e verificar se esse nível lógico aparece nas linhas (ou vice-versa). Se, por exemplo, a varredura for feita nas colunas e houver alteração no sinal lógico de alguma linha, significa que alguma tecla foi pressionada e, então, com base na coluna habilitada, sabe-se qual tecla foi pressionada. Nesse tipo de varredura, o emprego de resistores de *pull-up* ou *pull-down* nas vias de entrada é fundamental, visto que para a leitura as entradas sempre devem estar em um nível lógico conhecido. Na fig. 8.2, dois teclados com resistores de *pull-up* e *pull-down* são ilustrados, onde o sinal de varredura é aplicado às colunas, as saídas do sistema de controle.

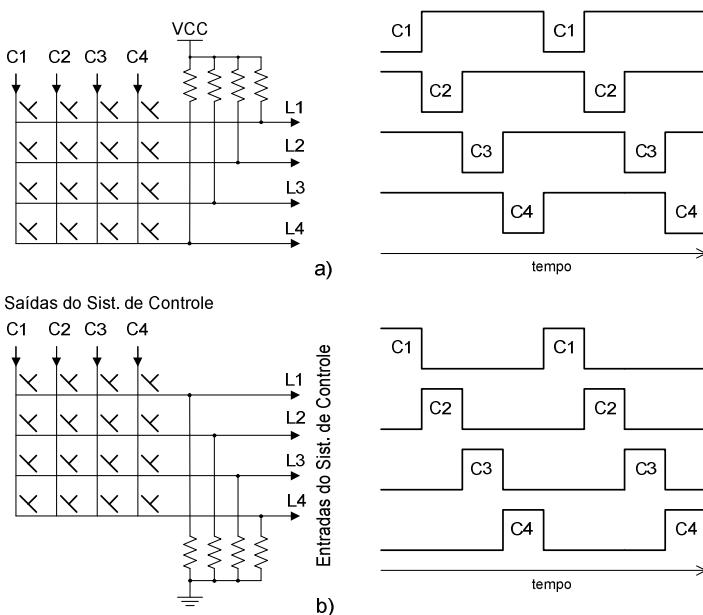


Fig. 8.2 – Teclados com resistores de *pull-up* (a) e *pull-down* (b) para as entradas do sistema de controle. A varredura é feita nas colunas.

No ATmega, a conexão de um teclado é facilmente obtida, pois existem resistores de *pull-up* habilitáveis em todos os pinos de I/O. Assim, um possível circuito para o trabalho com um teclado é apresentado na fig. 8.3. Um código exemplo com uma função para leitura desse teclado é apresentado na sequência.

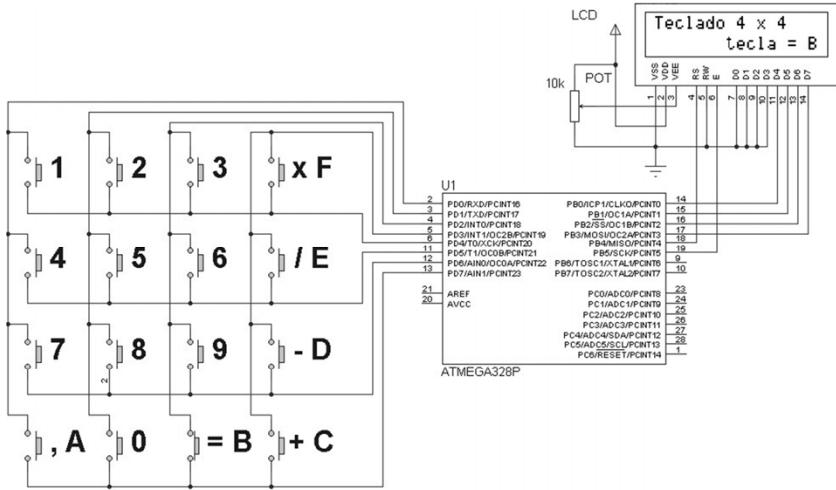


Fig. 8.3 – Teclado 4 × 4 controlado pelo ATmega328.

Teclado_hexa.c (programa principal)

```
===== //  
//          LEITURA DE UM TECLADO 4 x 4                      //  
===== //  
#include "def_principais.h"    //inclusão do arquivo com as principais definições  
#include "LCD.h"  
#include "teclado.h"  
  
//definição para acessar a memória flash como ponteiro  
prog_char mensagem1[] = "Teclado 4 x 4\0"; //mensagem armazenada na memória flash  
prog_char mensagem2[] = "tecla =\0";      //mensagem armazenada na memória flash  
  
int main()  
{  
    unsigned char nr;  
  
    DDRB = 0xFF;           //LCD esta no PORTB  
    DDRD = 0x0F;           //definições das entradas e saídas para o teclado  
    PORTD= 0xFF;           //habilita os pull-ups do PORTD e coloca colunas em 1  
    UCSR0B = 0x00;          //para uso dos PORTD no Arduino  
  
    inic_LCD_4bits();  
    escreve_LCD_Flash(mensagem1);  
    cmd_LCD(0xC7,0);      //desloca cursor para a 2a linha do LCD  
    escreve_LCD_Flash(mensagem2);
```

```

while(1)
{
    nr = ler_teclado(); //lê constantemente o teclado
    if(nr!=0xFF)//se alguma tecla foi pressionada mostra seu valor
    {
        cmd_LCD(0xCF,0);//desloca cursor para a última posição da 2a linha
        cmd_LCD(nr,1); //nr já está em formato ASCII
    }
}
//=====================================================================

```

teclado.h (arquivo de cabeçalho do teclado.c)

```

#ifndef _TECLADO_H
#define _TECLADO_H
#include "def_principais.h"

#define LINHA      PIND      //registrador para a leitura das linhas
#define COLUNA     PORTD     //registrador para a escrita nas colunas

//protótipo da função
unsigned char ler_teclado();

#endif

```

teclado.c (arquivo com a função de trabalho para o teclado)

```

//=====================================================================
Sub-rotina para o trabalho com um teclado com 16 teclas (4 colunas e 4 linhas)
organizados como:
    C1 C2 C3 C4
    x  x  x  x   L1
    x  x  x  x   L2
    x  x  x  x   L3
    x  x  x  x   L4
onde se deve empregar um único PORT conectado da seguinte maneira:
PORT = L4 L3 L2 L1 C4 C3 C2 C1 (sendo o LSB o C1 e o MSB o L4)
//===================================================================== */
#include "teclado.h"

/* matriz com as informações para decodificação do teclado,
organizada de acordo com a configuração do teclado, o usuário
pode definir valores números ou caracteres ASCII, como neste exemplo*/
const unsigned char teclado[4][4] PROGMEM = {{'1', '2', '3', 'F'},
                                             {'4', '5', '6', 'E'},
                                             {'7', '8', '9', 'D'},
                                             {'A', '0', 'B', 'C}};
//=====================================================================

```

```

unsigned char ler_teclado()
{
    unsigned char n, j, tecla=0xFF, linha;
    for(n=0;n<4;n++)
    {
        clr_bit(COLUNA,n); //apaga o bit da coluna (varredura)
        _delay_ms(10); /*atraso para uma varredura mais lenta, também elimina
                        o ruído da tecla*/
        linha = LINHA >> 4; //lê o valor das linhas
        for(j=0;j<4;j++) //testa as linhas
        {
            if(!tst_bit(linha,j))//se foi pressionada alguma tecla,
            { //decodifica e retorna o valor
                tecla = pgm_read_byte(&teclado[j][n]);
                /*while(!tst_bit(LINHA>>4,j));/*para esperar soltar a tecla, caso
                desejado, descomentar essa linha*/
            }
        }
        set_bit(COLUNA,n); //ativa o bit zerado anteriormente
    }
    return tecla; //retorna o valor 0xFF se nenhuma tecla foi pressionada
}
//-----

```

Os arquivos **def_principais.h**, **LCD.h** e **LCD.c** foram apresentados no capítulo 5 (LCD 16×2 com via de dados de 4 bits). Houve alteração do arquivo **LCD.h** para se adequar a nova configuração das conexões do LCD, resultando em:

```

#define DADOS_LCD      PORTB
#define nibble_dados   0
#define CONTR_LCD       PORTB
#define E               PB5
#define RS              PB4

```

O programa apresentado lê constantemente o teclado e apresenta o valor da tecla pressionada no LCD, a tecla B no caso exemplo da fig. 8.3. A função **ler_teclado()** emprega dois laços **for**, o mais externo para realizar a varredura das colunas, feita pela macro **clr_bit(COLUNA,n)**, cuja variável **n** determina qual coluna será colocada em zero. Na sequência, tem-se o tempo da varredura por coluna. O laço **for** interno verifica, para cada coluna selecionada, se alguma linha foi ativa; em caso afirmativo, o valor da tecla correspondente é atribuído para a variável **tecla** de acordo com a organização do teclado definido na variável **teclado[4][4]**.

Quando se faz a varredura do teclado, é importante utilizar uma frequência adequada. Se ela for muito alta, podem aparecer ruídos

espúrios; se for baixa, a tecla pode ser pressionada e solta sem que o sistema detecte o seu acionamento. Na função **ler_teclado()** foi utilizado um tempo de 10 ms para a varredura. Dessa forma, pela estruturação da função, é consumido um tempo aproximado de 40 ms na sua execução. Considerando-se que o ruído produzido pelo botão tem duração em torno de 10 ms, o tempo de execução da função poderia ser reduzido quatro vezes.

Exercícios:

- 8.1** – Elaborar um programa para um controle de acesso por senha numérica. A senha pode conter 3 dígitos. Toda vez que uma tecla for pressionada, um pequeno alto-falante deve ser acionado. Quando a senha for correta, o relé deve ser ligado por um pequeno tempo (utilize um LED de sinalização). Preveja que a senha possa ser alterada e salva na EEPROM. O circuito abaixo exemplifica o hardware de controle.

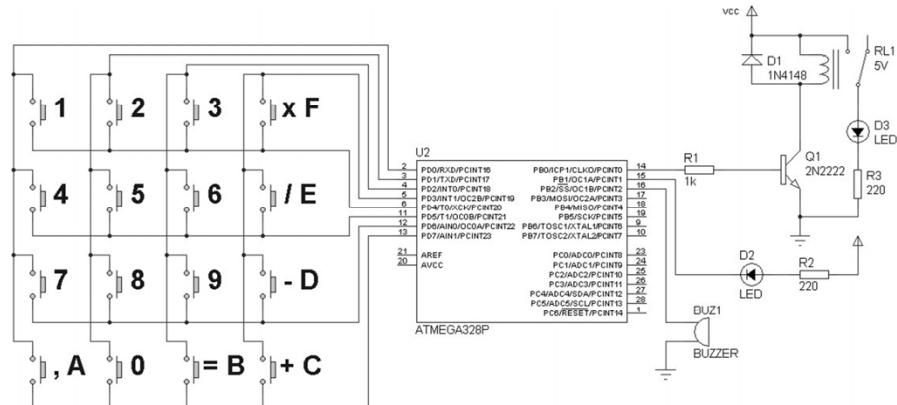


Fig. 8.4 – Controle de acesso por senha numérica.

- 8.2** – Elaborar um programa para ler um teclado alfanumérico, similar ao usado nos telefones celulares.
- 8.3** – Elaborar um programa para executar as funções matemáticas básicas de uma calculadora (números inteiros), conforme circuito da fig. 8.3. Consulte o apêndice B para a tabela de instruções do LCD, mude o sentido de deslocamento da mensagem para a esquerda na entrada de um novo caractere (0x07) e comece a escrita dos caracteres na última coluna da primeira linha.

9. TEMPORIZADORES/CONTADORES

Uma necessidade importante no trabalho com circuitos microcontrolados é a geração de sinais periódicos e eventos. Essas funcionalidades são realizadas pelos contadores/temporizadores (TCs) e muitos projetos só podem ser executados com o uso deles. Logo, entender o seu funcionamento é indispensável, permitindo a escrita de programas eficientes e projetos bem feitos. Neste capítulo, são apresentados os conceitos essenciais para o trabalho com os TCs do ATmega e um resumo para o trabalho com os seus registradores.

No ATmega328, existem dois temporizadores/contadores de 8 bits (TC0 e TC2) e um de 16 bits (TC1). Todos independentes e com características próprias.

O Temporizador/Contador 0 (TC0) emprega 8 bits (permite contagens de 0 até 255). Suas principais características são:

- Contador simples (baseado no *clock* da CPU).
- Contador de eventos externos.
- Divisor do *clock* para o contador de até 10 bits - *prescaler*.
- Gerador para 2 sinais PWM (pinos OC0A e OC0B).
- Gerador de frequência (onda quadrada).
- 3 fontes independentes de interrupção (por estouro e igualdades de comparação).

O Temporizador/Contador 2 (TC2) também emprega 8 bits e suas características são similares ao TC0. Entretanto, apresenta uma função especial para a contagem precisa de 1 s, permitindo o uso de um cristal externo independente para o seu *clock* (32,768 kHz). Pode gerar dois sinais PWM nos pinos OC2A e OC2B.

O Temporizador/Contador 1 (TC1) possui 16 bits e pode contar até 65535. Permite a execução precisa de temporizações, geração de formas de onda e medição de períodos de tempo. Suas principais características são:

- Contador simples (baseado no *clock* da CPU).
- Contador de eventos externos.
- Divisor do *clock* para o contador de até 10 bits - *prescaler*.
- Gerador para 2 sinais PWM (pinos OC1A e OC1B) com inúmeras possibilidades de configuração.
- Gerador de frequência (onda quadrada).
- 4 fontes independentes de interrupção (por estouro e igualdades de comparação).

9.1 TEMPORIZANDO E CONTANDO

Uma das principais função dos TCs é a contagem de pulsos de *clock*, os quais podem ser externos ou internos ao microcontrolador. Dessa forma, permitem a geração de eventos periódicos para uso do programador ou para a determinação de um número de contagens.

Um estouro do contador ocorre quando ele passa do valor máximo permitido para a contagem para o valor zero. Assim, se o TC for de 8 bits, ele contará de 0 até 255 resultando em 256 contagens; se for de 16 bits, contará de 0 até 65535, resultando em 65536 contagens. Assim, o tempo que um TC leva para estourar é dado por:

$$t_{estouro} = \frac{(TOP+1) \times prescaler}{f_{osc}} \quad (9.1)$$

onde *TOP* é o valor máximo de contagem, *f_{osc}* é a frequência do *clock* empregado (oscilador) e o *prescaler* é o divisor dessa frequência.

Exemplos:

1. Supondo um TC de 8 bits (conta até o valor TOP de 255), trabalhando com uma frequência de 1 MHz, sem divisor de frequência, o tempo para o seu estouro é de:

$$t_{estouro} = 256 \times \frac{1}{1 \text{ MHz}} \times 1 = 256 \mu\text{s}$$

2. Supondo um TC de 16 bits (conta até o valor TOP de 65535), trabalhando com uma frequência de 16 MHz, com divisor de frequência de 64, o tempo para o seu estouro é:

$$t_{estouro} = 65536 \times \frac{1}{16 \text{ MHz}} \times 64 = 0,262 \text{ s}$$

Na fig. 9.1, o funcionamento de um TC para o ATmega é ilustrado.

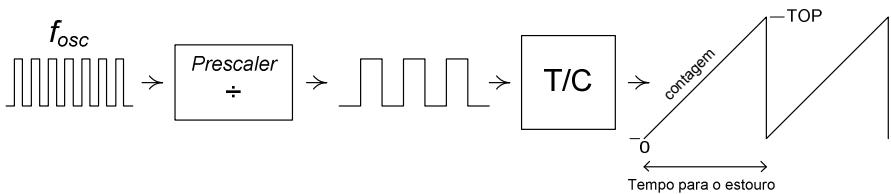


Fig. 9.1 – Funcionamento de um temporizador/contador do ATmega.

9.2 MODULAÇÃO POR LARGURA DE PULSO – PWM

A geração de sinais PWM é outra função importante dos TCs. O uso desses sinais é baseado no conceito de valor médio de uma forma de onda periódica. Baseado no valor médio de um sinal PWM, muitos dispositivos eletrônicos podem ser controlados, como por exemplo: motores, lâmpadas, LEDs, fontes chaveadas e circuitos inversores.

Digitalmente, o valor médio de uma forma de onda é controlado pelo tempo em que o sinal fica em nível lógico alto durante um determinado intervalo de tempo. No PWM, esse tempo é chamado de ciclo ativo (*Duty Cycle*). Na fig. 9.2, são apresentadas formas de onda PWM em que a largura do pulso (*Duty Cycle*) é variável de 0 até 100%, com incrementos de 25%.

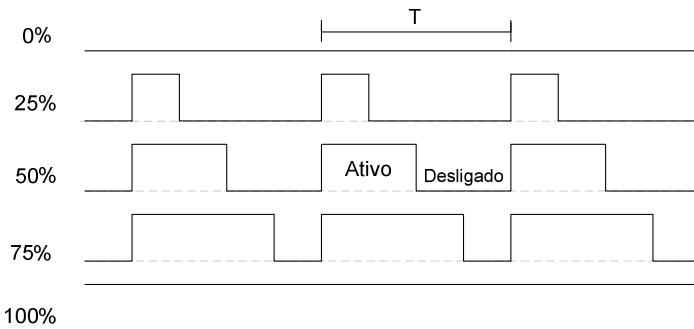


Fig. 9.2 – PWM com período T e ciclo ativo de 0%, 25%, 50%, 75% e 100%.

O cálculo do valor médio de um sinal digital é dado por:

$$V_{\text{médio}} = \frac{\text{Amplitude Máxima}}{\text{Período}} \times (\text{Tempo Ativo no Período}) \quad (9.2)$$

Assim, se o sinal digital tem variação de 0 a 5 V, um ciclo ativo de 50% corresponde a um valor médio de 2,5 V, enquanto um ciclo ativo de 75% corresponderia a 3,75 V. Logo, ao se alterar o ciclo útil do sinal PWM, altera-se o seu valor médio. Na fig. 9.3, é ilustrada a variação do ciclo ativo de um sinal PWM de 0 a 100% do seu ciclo útil (período) com o passar do tempo.

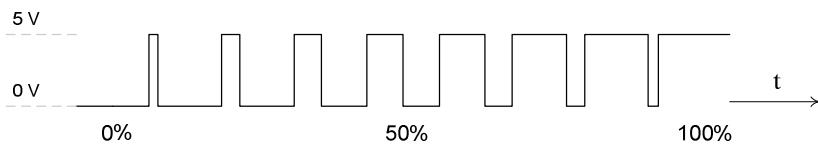


Fig. 9.3 – Variação do ciclo ativo de um sinal PWM em um intervalo de tempo.

É importante notar que o período do sinal PWM não se altera, e sim sua largura de ciclo ativo. Desta forma, quando se gera um sinal PWM, deve-se em primeiro lugar determinar sua frequência de trabalho.

A resolução do PWM em um microcontrolador é dada pelo seu número de bits e indica quantos diferentes ciclos ativos podem ser gerados. Por exemplo, um PWM de 10 bits pode gerar 1024 diferentes níveis, começando com o nível de tensão 0 e terminando com 100 % do ciclo ativo. Como será visto posteriormente, no ATmega o ciclo ativo de um sinal PWM é controlado pela escrita em registradores específicos e sua resolução vai depender do TC empregado.

Além de permitir a geração de sinais PWM, os TCs do ATmega permitem gerar a forma de onda quadrada. Sempre que se desejar gerar tais sinais, os pinos correspondentes do microcontrolador devem ser configurados como pinos de saída, caso contrário o sinal não aparecerá no pino.

9.3 TEMPORIZADOR/CONTADOR 0

O TC0 é um contador de 8 bits que é incrementado com pulsos de *clock*, os quais podem ser obtidos da fonte interna de *clock* do microcontrolador ou de uma fonte externa. Existem vários modos de operação: normal, CTC, PWM rápido e PWM com fase corrigida; permitindo desde simples contagens até a geração de diferentes tipos de sinais PWM.

MODO NORMAL

É o modo mais simples de operação, onde o TC0 conta continuamente de forma crescente. A contagem se dá de 0 até 255 voltando a 0, num ciclo contínuo. Quando a contagem passa de 255 para 0, ocorre o estouro e então, o bit sinalizador de estouro (TOV0) é colocado em 1. Se habilitada, uma interrupção é gerada.

A contagem é feita no registrador TCNT0 e um novo valor de contagem pode ser escrito a qualquer momento, permitindo-se alterar o número de contagens via programação.

MODO CTC

No modo CTC (*Clear Timer on Compare* - limpeza do contador na igualdade de comparação), o registrador OCR0A é usado para manipular a resolução do TC0. Neste modo, o contador é zerado quando o valor do contador (TCNT0) é igual ao valor de OCR0A (o valor TOP da contagem), ou seja, o contador conta de 0 até o valor de OCR0A. Isso permite um controle mais fino da frequência de operação e da resolução do TCO. O modo CTC permite configurar os pinos OC0A e OC0B para gerar ondas quadradas. Uma interrupção pode ser gerada cada vez que o contador atinge o valor de comparação (OCR0A ou OCR0B). Um exemplo de diagrama de tempo para o modo CTC é mostrado na fig. 9.4, onde os pinos OC0A e OC0B foram configurados para trocar de estado quando ocorre a igualdade entre o valor do registrador TCNT0 com OCR0A e OCR0B.

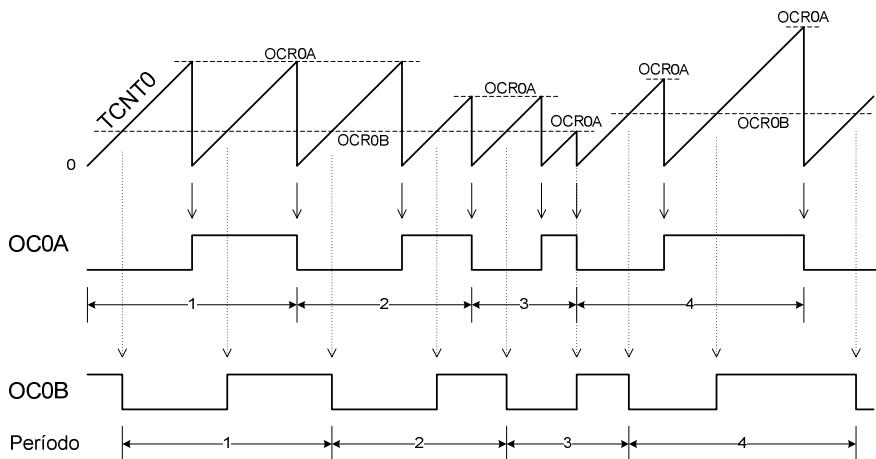


Fig. 9.4 – Modo CTC para a geração de ondas quadradas.

No exemplo, para o pino OC0B, o registrador OCR0B é empregado para deslocar a forma de onda em relação a OC0A e deve ter valor menor ou igual a OCR0A. Para gerar uma onda de saída no modo CTC, os pinos OC0A e/ou OC0B devem ser configurados como saídas e ajustados para

trocar de estado em cada igualdade de comparação através dos bits do modo de comparação.

A frequência no modo CTC é definida por:

$$f_{OC0A} = \frac{f_{osc}}{2N(1+OCR0A)} \quad [\text{Hz}] \quad (9.3)$$

onde f_{osc} é a frequência de operação do microcontrolador, OCR0A assume valores entre 0 e 255 e N é o fator do *prescaler* (1, 8, 64, 256 ou 1024). Para calcular o valor de OCR0A baseado na frequência desejada, basta isolá-lo na equação acima, o que resulta em:

$$OCR0A = \frac{f_{osc}}{2N \cdot f_{OC0A}} - 1 \quad (9.4)$$

O modo CTC não necessita ser empregado para a geração de formas de onda, pode ser utilizado para temporizações e contagens externas. Ele permite flexibilizar o número de contagens do TC0 e gerar interrupções por igualdade de comparação do TCNT0 com os registradores OCR0A e OCR0B.

MODO PWM RÁPIDO

O modo de modulação por largura de pulso rápido permite a geração de um sinal PWM de alta frequência. O contador conta de zero até o valor máximo e volta a zero. No modo de comparação com saída não-invertida, o pino OC0A é zerado na igualdade entre TCNT0 e OCR0A, e colocado em 1 no valor mínimo do contador. No modo de comparação com saída invertida, o processo é inverso: OC0A é ativo na igualdade e zerado no valor mínimo. É o registrador OCR0A que determina o ciclo ativo do sinal PWM. Também é possível habilitar o pino OC0B para gerar um sinal PWM, onde o ciclo ativo do PWM será controlado pelo valor de OCR0B. A contagem do TC0 pode ser ajustada para 255 ou pelo valor dado por OCR0A; se for utilizado o registrador OCR0A, o pino OC0A não poderá gerar um sinal PWM, apenas uma onda quadrada. Quando se controla o ciclo ativo do sinal

PWM, o valor zero para OCR0A produz um pequeno ruído e o valor máximo (255) vai deixar o sinal PWM em 0 ou 1, conforme foi habilitada a saída, invertida ou não.

A alta frequência do PWM rápido o torna adequado para aplicações de regulação de potência, retificação e outras usando conversores DAs. Na fig. 9.5, é apresentado o exemplo do diagrama temporal para o modo PWM rápido para as saídas não invertidas e contagem máxima do TC de 255.

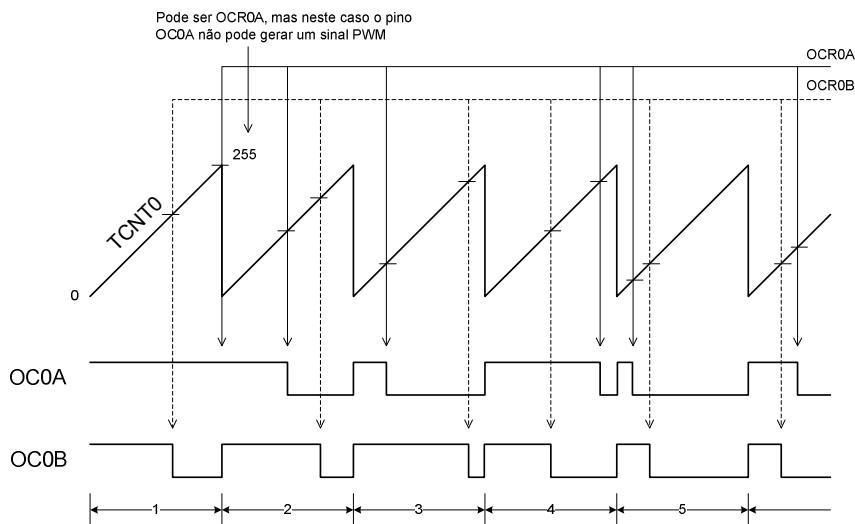


Fig. 9.5 – Modo PWM rápido, saídas não invertidas.

O valor de comparação OCR0A pode ser atualizado na interrupção por estouro do contador cada vez que o contador chegar no valor máximo de contagem, gerando um sinal PWM variável.

A frequência de saída do PWM é calculada como:

$$f_{OC0A_PWM} = \frac{f_{osc}}{N(1+TOP)} \quad [\text{Hz}] \quad (9.5)$$

onde f_{osc} é a frequência de operação do microcontrolador, N é o fator do *prescaler* (1, 8, 64, 256 ou 1024) e TOP assume valores conforme tab. 9.7.

Um sinal com ciclo ativo do PWM de 50% pode ser obtido ajustando-se OC0A para mudar de estado a cada igualdade de comparação. A máxima frequência é obtida quando OCR0A é zero ($f_{OC2_PWM} = f_{osc}/2$).

MODO PWM COM FASE CORRIGIDA

O modo PWM com fase corrigida permite ajustar a fase do PWM, isto é, o início e fim do ciclo ativo do sinal PWM. É baseado na contagem crescente e decrescente do TCNT0, que conta repetidamente de zero até o valor máximo (TOP) e vice-versa (permanece um ciclo de *clock* no valor TOP), o que torna a saída PWM simétrica dentro de um período. Além disso, o valor de comparação que determina o razão cíclica da saída PWM é armazenado em registradores, sendo atualizado apenas quando o contador atinge o valor TOP, o qual marca o início e o fim de um ciclo PWM. Assim, se o valor TOP não for alterado com o temporizador em operação, o pulso gerado será sempre simétrico em relação ao ponto médio do período (TCNT0 = 0x00), qualquer que seja a razão cíclica.

Por ser um sinal que permite um maior controle do ciclo ativo do sinal PWM, é adequado para o controle de motores, sendo mais lento e preciso que o modo PWM rápido.

Para o sinal PWM não invertido, o pino OC0A é zerado na igualdade entre TCNT0 e OCR0A, quando a contagem é crescente, e colocado em 1 quando a contagem é decrescente. No modo de comparação com saída invertida, o processo é contrário. A comparação para o pino OC0B é feita com o registrador OCR0B. O valor máximo de contagem pode ser definido como 255 ou pelo valor de OCR0A, como no modo PWM rápido; se OCR0A for utilizado para determinar o valor TOP, o pino OC0A não poderá gerar um sinal PWM, somente uma onda quadrada.

Na fig. 9.6, é exemplificado um diagrama de tempo do modo PWM com fase corrigida com a saída OC0A não invertida e a saída OC0B invertida.

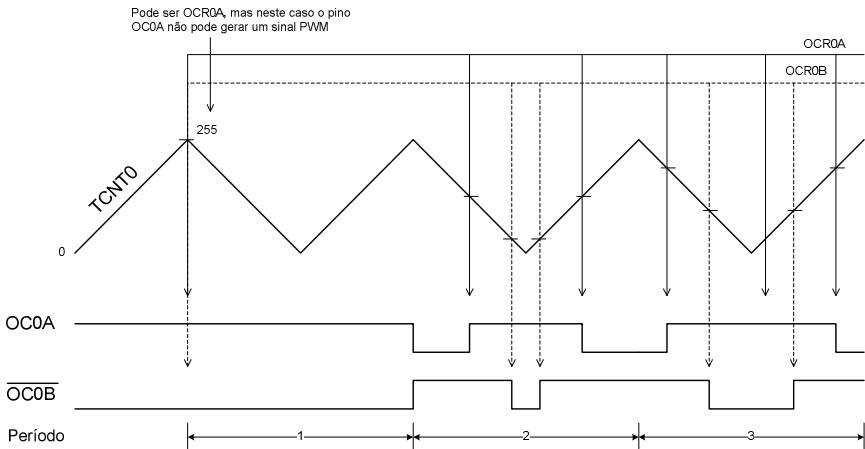


Fig. 9.6 – Modo PWM com correção de fase , OC0A não invertida e OC0B invertida.

A frequência de saída do PWM com fase corrigida é calculada como:

$$f_{OC0A_PWM} = \frac{f_{osc}}{2N(1+TOP)} \quad [\text{Hz}] \quad (9.6)$$

onde f_{osc} é a frequência de operação do microcontrolador, N é o fator do *prescaler* (1, 8, 64, 256 ou 1024) e TOP assume valores conforme tab. 9.7.

9.3.1 REGISTRADORES DO TCO

O controle do modo de operação do TCO é feito nos registradores TCCR0A e TCCR0B (*Timer/Counter Control 0 Register A e B*).

Bit	7	6	5	4	3	2	1	0
TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
Lê/Escr.	L/E	L/E	L/E	L/E	L	L	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

Bits 7:6 – COM0A1:0 – Compare Match Output A Mode

Estes bits controlam o comportamento do pino OC0A (*Output Compare 0A*). Se um ou ambos os bits forem colocados em 1, a funcionalidade normal do pino é alterada. Entretanto, o bit do registrador DDRx correspondente ao pino OC0A deve estar ajustado para habilitar o *driver* de saída. A funcionalidade dos bits COM0A1:0 depende do ajuste dos bits WGM02:0. Suas possíveis configurações são apresentadas nas tabs. 9.1-3.

Tab. 9.1 – Modo CTC (não PWM).

COM0A1	COM0A0	Descrição
0	0	Operação normal do pino, OC0A desconectado.
0	1	Mudança do estado de OC0A na igualdade de comparação.
1	0	OC0A é limpo na igualdade de comparação.
1	1	OC0A é ativo na igualdade de comparação.

Tab. 9.2 – Modo PWM rápido.

COM0A1	COM0A0	Descrição
0	0	Operação normal do pino, OC0A desconectado.
0	1	WGM02 = 0: operação normal do pino, OC0A desconectado. WGM02 = 1: troca de estado do OC0A na igualdade de comparação.
1	0	OC0A é limpo na igualdade de comparação, OC0A ativo no valor do TC mínimo (modo não invertido).
1	1	OC0A é ativo na igualdade de comparação e limpo no valor do TC mínimo (modo invertido).

Tab. 9.3 – Modo PWM com fase corrigida.

COM0A1	COM0A0	Descrição
0	0	Operação normal do pino, OC0A desconectado.
0	1	WGM02 = 0: operação normal do pino, OC0A desconectado. WGM02 = 1: troca de estado do OC0A na igualdade de comparação.
1	0	OC0A é limpo é na igualdade de comparação quando a contagem é crescente, e ativo na igualdade de comparação quando a contagem é decrescente.
1	1	OC0A é ativo na igualdade de comparação quando a contagem é crescente, e limpo na igualdade de comparação quando a contagem é decrescente.

Bits 5:4 – COM0B1:0 – Compare Match Output B Mode

Estes bits controlam o comportamento do pino OC0B (*Output Compare 0B*). Se um ou ambos os bits forem colocados em 1, a funcionalidade normal do pino é alterada. Entretanto, o bit do registrador DDRx correspondente ao pino OC0B deve estar ajustado para habilitar o *driver* de saída. A funcionalidade dos bits COM0B1:0 depende do ajuste dos bits WGM02:0. Suas possíveis configurações são apresentadas nas tabs. 9.4-6.

Tab. 9.4 – Modo CTC (não PWM).

COM0B1	COM0B0	Descrição
0	0	Operação normal do pino, OC0B desconectado.
0	1	Mudança do estado de OC0B na igualdade de comparação.
1	0	OC0B é limpo na igualdade de comparação.
1	1	OC0B é ativo na igualdade de comparação.

Tab. 9.5 – Modo PWM rápido.

COM0B1	COM0B0	Descrição
0	0	Operação normal do pino, OC0B desconectado.
0	1	Reservado.
1	0	OC0B é limpo na igualdade de comparação, OC0B ativo no valor do TC mínimo (modo não invertido).
1	1	OC0B é ativo na igualdade de comparação e limpo no valor do TC mínimo (modo invertido).

Tab. 9.6 – Modo PWM com fase corrigida.

COM0B1	COM0B0	Descrição
0	0	Operação normal do pino, OC0B desconectado.
0	1	Reservado.
1	0	OC0B é limpo é na igualdade de comparação quando a contagem é crescente, e ativo na igualdade de comparação quando a contagem é decrescente.
1	1	OC0B é ativo na igualdade de comparação quando a contagem é crescente, e limpo na igualdade de comparação quando a contagem é decrescente.

Bits 1:0 – WGM01:0 – Wave Form Generation Mode

Combinados com o bit WGM02 do registrador TCCROB, estes bits controlam a sequência de contagem do contador, a fonte do valor máximo para contagem (TOP) e o tipo de forma de onda a ser gerada, conforme tab. 9.7.

Tab. 9.7 – Bits para configurar o modo de operação do TCO.

Modo	WGM02	WGM01	WGM00	Modo de Operação TC	TOP	Atualização de OCR0A no valor:	Sinalização do bit TOV0 no valor:
0	0	0	0	Normal	0xFF	Imediata	0xFF
1	0	0	1	PWM com fase corrigida	0xFF	0xFF	0x00
2	0	1	0	CTC	OCR0A	Imediata	0xFF
3	0	1	1	PWM rápido	0xFF	0x00	0xFF
4	1	0	0	Reservado	-	-	-
5	1	0	1	PWM com fase corrigida	OCR0A	OCR0A	0x00
6	1	1	0	Reservado	-	-	-
7	1	1	1	PWM rápido	OCR0A	0x00	OCR0A

TCCR0B – Timer/Counter 0 Control Register B

Bit	7	6	5	4	3	2	1	0	
	TCCR0B	FOCOA	FOCOB	-	-	WGM02	CS02	CS01	CS00
Lê/Escr.	E	E	L	L	L/E	L/E	L/E	L/E	
Valor Inic.	0	0	0	0	0	0	0	0	

Bits 7:6 – FOC0A:B – Force Output Compare A e B

Estes bits são ativos somente para os modos não-PWM. Quando em 1, uma comparação é forçada no módulo gerador de onda. O efeito nas saídas dependerá da configuração dada aos bits COM0A1:0 e COM0B1:0.

Bit 3 – WGM02 – Wave Form Generation Mode

Função descrita na tab. 9.7.

Bits 2:0 – CS02:0 – Clock Select

Bits para seleção da fonte de *clock* para o TC0, conforme tab. 9.8.

Tab. 9.8 – Seleção do *clock* para o TC0.

CS02	CS01	CS00	Descrição
0	0	0	Sem fonte de <i>clock</i> (TC0 parado).
0	0	1	<i>clock</i> /1 (<i>prescaler</i> =1) - sem <i>prescaler</i> .
0	1	0	<i>clock</i> /8 (<i>prescaler</i> = 8).
0	1	1	<i>clock</i> /64 (<i>prescaler</i> = 64).
1	0	0	<i>clock</i> /256 (<i>prescaler</i> = 256).
1	0	1	<i>clock</i> /1024 (<i>prescaler</i> = 1024).
1	1	0	<i>clock</i> externo no pino T0. Contagem na borda de descida.
1	1	1	<i>clock</i> externo no pino T0. Contagem na borda de subida.

TCNT0 – Timer/Counter 0 Register

Registrador de 8 bits onde é realizada a contagem do TC0, pode ser lido ou escrito a qualquer tempo.

OCROA – Output Compare 0 Register A

Registrador de comparação A de 8 bits, possui o valor que é continuamente comparado com o valor do contador (TCNT0). A igualdade pode ser utilizada para gerar uma interrupção ou uma forma de onda no pino OC0A.

OCROB – Output Compare 0 Register B

Registrador de comparação B de 8 bits, possui o valor que é continuamente comparado com o valor do contador (TCNT0). A igualdade pode ser utilizada para gerar uma interrupção ou uma forma de onda no pino OC0B.

TIMSK0 – Timer/Counter 0 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
TIMSK0	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 2 – OCIE0B – Timer/Counter 0 Output Compare Match B Interrupt Enable

A escrita 1 neste bit ativa a interrupção do TC0 na igualdade de comparação com o registrador OCR0B.

Bit 1 – OCIE0A – Timer/Counter 0 Output Compare Match A Interrupt Enable

A escrita 1 neste bit ativa a interrupção do TC0 na igualdade de comparação com o registrador OCR0A.

Bit 0 – TOIE0 – Timer/Counter 0 Overflow Interrupt Enable

A escrita 1 neste bit ativa a interrupção por estouro do TC0.

As interrupções individuais dependem da habilitação das interrupções globais pelo bit I do SREG.

TIFR0 – Timer/Counter 0 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
TIFR0	-	-	-	-	-	OCF0B	OCF0A	TOV0
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 2 – OCF0B – Timer/Counter 0 Output Compare Match B Match Flag

Este bit é colocado em 1 quando o valor da contagem (TCNT0) é igual ao valor do registrador de comparação de saída B (OCR0B) do TC0.

Bit 1 – OCF0A – Timer/Counter 0 Output Compare Match A Match Flag

Este bit é colocado em 1 quando o valor da contagem (TCNT0) é igual ao valor do registrador de comparação de saída A (OCR0A) do TC0.

Bit 0 – TOV0 – Timer/Counter 0 Overflow Flag

Este bit é colocado em 1 quando um estouro do TC0 ocorre.

9.3.2 CÓDIGOS EXEMPLO

MODO NORMAL – GERANDO UM PEDIDO DE INTERRUPÇÃO

O código abaixo é utilizado para gerar um evento periódico de interrupção a cada 16,384 ms. Neste caso, um LED ligado ao pino PB5 troca de estado (liga ou desliga) a cada interrupção. Foi empregado o maior divisor de *clock* possível, 1024, e não há possibilidade de gerar um intervalo de tempo maior sem a diminuição da frequência de trabalho do microcontrolador.

TC0_estouro.c

```
//==============================================================================
// HABILITANDO A INTERRUPÇÃO POR ESTOURO DO TC0
//==============================================================================
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>

#define cpl_bit(y,bit) (y^=(1<<bit))//troca o estado lógico do bit x da variável Y
#define LED      PB5

ISR(TIMER0_OVF_vect) //interrupção do TC0
{
    cpl_bit(PORTB,LED);
}

int main()
{
    DDRB = 0b00010000;//somente pino do LED como saída
    PORTB = 0b11011111;//apaga LED e habilita pull-ups nos pinos não utilizados

    TCCR0B = (1<<CS02) | (1<<CS00);/*TC0 com prescaler de 1024, a 16 MHz gera uma
                                         interrupção a cada 16,384 ms*/
    TIMSK0 = 1<<TOIE0;   //habilita a interrupção do TC0
    sei();                  //habilita a chave de interrupção global

    while(1)
    {
        /*Aqui vai o código, a cada estouro do TC0 o programa desvia para
         ISR(TIMER0_OVF_vect)*/
    }
}
//==============================================================================
```

MODO CTC

Neste exemplo, os pinos OC0A e OC0B são configurados para trocar de estado na igualdade de comparação do registrador TCNT0 com o OCR0A e OCR0B, respectivamente.

TC0_PWMs.c

```
#define F_CPU 16000000UL
#include <avr/io.h>

int main(void)
{
    DDRD = 0b01100000; //pinos OC0B e OC0A (PD5 e PD6) como saída
    PORTD = 0b10011111; //zera saídas e habilita pull-ups nos pinos não utilizados

    //MODO CTC
    TCCR0A = 0b01010010; /*habilita OC0A e OC0B para trocar de estado na igualdade de
                           comparação*/
    TCCR0B = 0b00000001; //liga TC0 com prescaler = 1.
    OCR0A = 200;          //máximo valor de contagem
    OCR0B = 100;           //deslocamento de OC0B em relação a OC0A

    while(1)
    {
        //o programa principal vai aqui
    }
}
```

MODO PWM RÁPIDO

Abaixo são apresentadas algumas configurações, considerando que os pinos OC0A e OC0B estejam configurados como saída.

```
...
/*fast PWM, TOP = 0xFF, OC0A e OC0B habilitados
TCCR0A = 0b10100011; //PWM não invertido nos pinos OC0A e OC0B
TCCR0B = 0b00000011; //liga TC0, prescaler = 64
OCR0A = 200;          //controle do ciclo ativo do PWM OC0A
OCR0B = 50;           //controle do ciclo ativo do PWM OC0B

...
/*fast PWM, TOP = OCR0A com OC0A gerando uma onda quadrada e OC0B um sinal PWM não
                           Invertido*/
TCCR0A = 0b01100011; //TOP = OCR0A, OC0A e OC0B habilitados
TCCR0B = 0b00001001; //liga TC0, prescaler = 1 e ajusta modo para comparação com OCR0A
OCR0A = 100;          //controle da período do sinal no pino OC0A
OCR0B = 30;           //controle do ciclo ativo do PWM OC0B
```

MODO PWM COM FASE CORRIGIDA

Abaixo são apresentadas algumas configurações, considerando que os pinos OC0A e OC0B estejam configurados como saída.

```
...
//phase correct PWM, TOP = 0xFF
TCCR0A = 0b10100001; //OC0A e OC0B habilitados
TCCR0B = 0b00000001; //liga TCO, prescaler = 1
OCR0A = 100;          //controle do período do sinal no pino OC0A
OCR0B = 50;           //controle do ciclo ativo do PWM 0B

...
//phase correct PWM, TOP = OCR0A
TCCR0A = 0b01100011; //OC0A e OC0B habilitado
TCCR0B = 0b00001001; /*liga TCO, prescaler = 1 e habilita o pino OC0A para gerar um
                     onda quadrada*/
OCR0A = 200;          //controle do período do sinal no pino OC0A
OCR0B = 10;            //controle do ciclo ativo do PWM OC0B
```

Exercícios:

9.1 – Considerando o TCO trabalhando a 8 MHz com um *prescaler* de 256, quanto tempo leva para o TCO gerar um estouro de contagem?

9.2 – Considerando-se o PWM rápido configurado para que o valor máximo de contagem do TCO seja dado pelo valor de OCR0A e que o pino OC0B esteja configurado para gerar o sinal PWM, determine:

- Qual o período do sinal PWM se o ATmega328 estiver rodando a 12 MHz e o valor de OCR0A for 99?
- Supondo que OCR0A é 200, qual deve ser o valor de OCR0B para que o ciclo ativo do sinal PWM (OC0B) seja de 75 %?

9.3 – Verifique a configuração dada ao TCO para os exemplos acima. Analise os bits dos registradores envolvidos.

9.4 – Com o emprego de um osciloscópio, teste os sinais gerados pelos códigos exemplo dados previamente, gravando-os no Arduino. Altere os valores dos registradores OCR0A e OCR0B para analisar o comportamento do microcontrolador. Para a análise, também pode ser empregado o software de simulação Proteus (ISIS).

9.5 – Elaborar um programa para ler 6 teclas utilizando a interrupção externa INT0 do ATmega328 (ver o capítulo 6). O programa principal não deve ficar responsável pela varredura do teclado (ver o capítulo 8); a interrupção do TCO deve ser empregada para esse fim. O tratamento do

teclado será transparente ao programa principal, devendo ser feito nas interrupções. Assim, quando uma tecla for pressionada será requisitado um pedido de interrupção e o LED correspondente deve ser ligado. O circuito abaixo ilustra o circuito necessário.

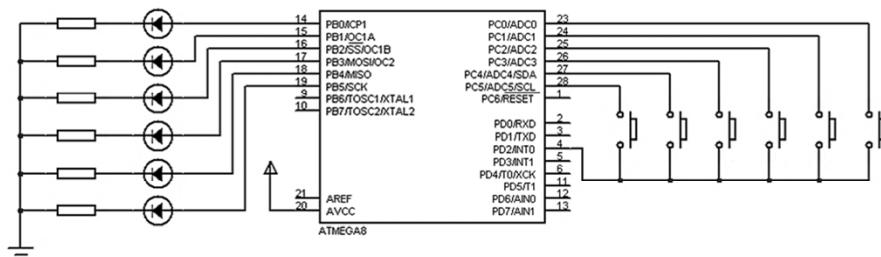


Fig. 9.7 – Seis teclas gerando um pedido de interrupção com o emprego de apenas uma interrupção externa (obs.: a pinagem do ATmega8 é igual a do ATmega328).

9.4 TEMPORIZADOR/CONTADOR 2

O TC2 possui características similares a do TC0, é um contador de 8 bits que permite os modos normal, CTC, PWM rápido e PWM com fase corrigida. Possui dois pinos de saída para os sinais gerados, OC2A (PB3) e OC2B (PD3), ligados aos PORTB e PORTD, respectivamente. Não possui entrada para contagem de sinais de *clock* externos como o TC0. Entretanto, possui mais possibilidades para a divisão do *clock* para a contagem (*prescaler*) e uma característica peculiar: permite o uso de um cristal externo de 32,768 kHz para o seu circuito de *clock*. Essa característica é denominada operação assíncrona.

Para o TC2, os registradores de comparação são o OCR2A e OCR2B e as fórmulas de cálculo para as frequências de trabalho do PWM são similares às do TC0, ver as eqs. 9.3-6. O registrador principal é o OCR2A, que pode ser configurado para determinar o valor máximo de contagem. O *prescaler* pode assumir os seguintes valores: 1, 8, 32, 64, 128, 256 ou 1024.

OPERAÇÃO ASSÍNCRONA

Neste modo, o TC2 utiliza um cristal de relógio de 32,768 kHz para gerar precisamente uma base de tempo de 1 s ou frações dessa. O TC2 possui um circuito oscilador otimizado exclusivamente para o trabalho com esse cristal. Essa característica do TC2 é chamada pela Atmel de RTc – *Real Timer counter*.

Para o ATmega328, os pinos de conexão para o cristal de 32,768 kHz (TOSC1 e TOSC2) são os mesmos pinos de conexão para o uso de um cristal externo ou ressonador cerâmico (pinos XTAL1 e XTAL2) que definem a frequência de trabalho da CPU e, portanto, se for empregado um, o outro não pode ser conectado. Caso o modo assíncrono seja utilizado, é obrigatório o uso do oscilador interno para gerar o sinal de *clock* do microcontrolador. No Arduino não é possível utilizar a operação assíncrona do TC2, pois o microcontrolador trabalha com um cristal de 16 MHz soldado à placa do hardware.

9.4.1 REGISTRADORES DO TC2

O controle do modo de operação do TC2 é feito nos registradores TCCR2A e TCCR2B (*Timer/Counter Control 2 Register A e B*).

Bit	7	6	5	4	3	2	1	0
TCCR2A	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20
Lê/Escr.	L/E	L/E	L/E	L/E	L	L	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

Bits 7:6 – COM2A1:0 – Compare Match Output A Mode

Estes bits controlam o comportamento do pino OC2A (*Output Compare 2A*). Se um ou ambos os bits forem colocados em 1, a funcionalidade normal do pino é alterada. Entretanto, o bit do registrador DDRx correspondente ao pino OC2A deve estar ajustado para habilitar o *driver* de saída. A funcionalidade dos bits COM2A1:0 depende do ajuste dos bits WGM22:0. Configurações dadas nas tabs. 9.9-11.

Tab. 9.9 – Modo CTC (não PWM).

COM2A1	COM2A0	Descrição
0	0	Operação normal do pino, OC2A desconectado.
0	1	Mudança do estado de OC2A na igualdade de comparação.
1	0	OC2A é limpo na igualdade de comparação.
1	1	OC2A é ativo na igualdade de comparação.

Tab. 9.10 – Modo PWM rápido.

COM2A1	COM2A0	Descrição
0	0	Operação normal do pino, OC2A desconectado.
0	1	WGM22 = 0: operação normal do pino, OC2A desconectado. WGM22 = 1: troca de estado do OC2A na igualdade de comparação.
1	0	OC2A é limpo na igualdade de comparação, OC2A ativo no valor do TC mínimo (modo não invertido).
1	1	OC2A é ativo na igualdade de comparação e limpo no valor do TC mínimo (modo invertido).

Tab. 9.11 – Modo PWM com fase corrigida.

COM2A1	COM2A0	Descrição
0	0	Operação normal do pino, OC2A desconectado.
0	1	WGM22 = 0: operação normal do pino, OC2A desconectado. WGM22 = 1: troca de estado do OC2A na igualdade de comparação.
1	0	OC2A é limpo é na igualdade de comparação quando a contagem é crescente, e ativo na igualdade de comparação quando a contagem é decrescente.
1	1	OC2A é ativo na igualdade de comparação quando a contagem é crescente, e limpo na igualdade de comparação quando a contagem é decrescente.

Bits 5:4 – COM2B1:0 – Compare Match Output B Mode

Estes bits controlam o comportamento do pino OC2B (*Output Compare 2B*). Se um ou ambos os bits forem colocados em 1, a funcionalidade normal do pino é alterada. Entretanto, o bit do registrador DDRx correspondente ao pino OC2B deve estar ajustado para habilitar o *driver* de saída. A funcionalidade dos bits COM2B1:0 depende do ajuste dos bits WGM22:0. Suas possíveis configurações são dadas nas tabs. 9.12-14.

Tab. 9.12 – Modo CTC (não PWM).

COM2B1	COM2B0	Descrição
0	0	Operação normal do pino, OC2B desconectado.
0	1	Mudança do estado de OC2B na igualdade de comparação.
1	0	OC2B é limpo na igualdade de comparação.
1	1	OC2B é ativo na igualdade de comparação.

Tab. 9.13 – Modo PWM rápido.

COM2B1	COM2B0	Descrição
0	0	Operação normal do pino, OC2B desconectado.
0	1	Reservado.
1	0	OC2B é limpo na igualdade de comparação, OC2B ativo no valor do TC mínimo (modo não invertido).
1	1	OC2B é ativo na igualdade de comparação e limpo no valor do TC mínimo (modo invertido).

Tab. 9.14 – Modo PWM com fase corrigida.

COM2B1	COM2B0	Descrição
0	0	Operação normal do pino, OC2B desconectado.
0	1	Reservado
1	0	OC2B é limpo é na igualdade de comparação quando a contagem é crescente, e ativo na igualdade de comparação quando a contagem é decrescente.
1	1	OC2B é ativo na igualdade de comparação quando a contagem é crescente, e limpo na igualdade de comparação quando a contagem é decrescente.

Bits 1:0 – WGM21:0 – Wave Form Generation Mode

Combinados com o bit WGM22 do registrador TCCR2B, esses bits controlam a sequência de contagem do contador, a fonte do valor máximo para contagem (TOP) e o tipo de forma de onda a ser gerada, conforme tab. 9.15.

Tab. 9.15 – Bits para configurar o modo de operação do TC2.

Modo	WGM22	WGM21	WGM20	Modo de Operação TC	TOP	Atualização de OCR2A no valor:	Sinalização do bit TOV2 no valor:
0	0	0	0	Normal	0xFF	Imediata	0xFF
1	0	0	1	PWM com fase corrigida	0xFF	0xFF	0x00
2	0	1	0	CTC	OCR2A	Imediata	OCR2A
3	0	1	1	PWM rápido	0xFF	0x00	0xFF
4	1	0	0	Reservado	-	-	-
5	1	0	1	PWM com fase corrigida	OCR2A	OCR2A anterior	0x00
6	1	1	0	Reservado	-	-	-
7	1	1	1	PWM rápido	OCR2A	0x00	OCR2A

TCCR2B – Timer/Counter 2 Control Register B

Bit	7	6	5	4	3	2	1	0
TCCR2B	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20
Lê/Escr.	E	E	L	L	L/E	L/E	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

Bits 7:6 – FOC2A:B – Force Output Compare A e B

Estes bits são ativos somente para os modos não-PWM. Quando em 1, obrigam uma comparação no módulo gerador de forma de onda. O efeito nas saídas dependerá da configuração dado aos bits COM2A1:0 e COM2B1:0.

Bit 3 – WGM22 - Wave Form Generation Mode

Função descrita na tab. 9.15.

Bits 2:0 – CS22:0 – Clock Select

Bits para seleção da fonte de *clock* para o TC2, conforme tab. 9.16.

Tab. 9.16 – Seleção do *clock* para o TC2.

CS22	CS21	CS20	Descrição
0	0	0	Sem fonte de <i>clock</i> (TC2 parado)
0	0	1	<i>clock</i> /1 (<i>prescaler</i> =1) - sem <i>prescaler</i>
0	1	0	<i>clock</i> /8 (<i>prescaler</i> = 8)
0	1	1	<i>clock</i> /32 (<i>prescaler</i> = 32)
1	0	0	<i>clock</i> /64 (<i>prescaler</i> = 64)
1	0	1	<i>clock</i> /128 (<i>prescaler</i> = 128)
1	1	0	<i>clock</i> /256 (<i>prescaler</i> = 256)
1	1	1	<i>clock</i> /1024 (<i>prescaler</i> = 1024)

TCNT2 – Timer/Counter 2 Register

Registrador de 8 bits onde é realizada a contagem do TC2, pode ser lido ou escrito a qualquer momento.

OCR2A – Output Compare 2 Register A

Registrador de comparação A de 8 bits, possui o valor que é continuamente comparado com o valor do contador (TCNT2). A igualdade pode ser utilizada para gerar uma interrupção ou uma forma de onda no pino OC2A.

OCR2B – Output Compare 2 Register B

Registrador de comparação B de 8 bits, possui o valor que é continuamente comparado com o valor do contador (TCNT2). A igualdade pode ser utilizada para gerar uma interrupção ou uma forma de onda no pino OC2B.

TIMSK2 – Timer/Counter 2 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
TIMSK2	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 2 – OCIE2B – Timer/Counter 2 Output Compare Match B Interrupt Enable

A escrita de 1 neste bit ativa a interrupção do TC2 na igualdade de comparação com o registrador OCR2B.

Bit 1 – OCIE2A – Timer/Counter 2 Output Compare Match A Interrupt Enable

A escrita de 1 neste bit ativa a interrupção do TC2 na igualdade de comparação com o registrador OCR2A.

Bit 0 – TOIE2 – Timer/Counter 2, Overflow Interrupt Enable

A escrita de 1 neste bit ativa a interrupção por estouro do TC2.

As interrupções individuais dependem da habilitação das interrupções globais pelo bit I do registrador SREG.

TIFR2 – Timer/Counter 2 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
TIFR2	-	-	-	-	-	OCF2B	OCF2A	TOV2
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 2 – OCF2B – Timer/Counter 2, Output Compare B Match Flag

Este bit é colocado em 1 quando o valor da contagem (TCNT2) é igual ao valor do registrador de comparação de saída B (OCR2B) do TC2.

Bit 1 – OCF2A – Timer/Counter 2, Output Compare A Match Flag

Este bit é colocado em 1 quando o valor da contagem (TCNT2) é igual ao valor do registrador de comparação de saída A (OCR2A) do TC2.

Bit 0 – TOV2 – Timer/Counter 2 Overflow Flag

Este bit é colocado em 1 quando um estouro do TC2 ocorre.

ASSR – Asynchronous Status Register

Bit	7	6	5	4	3	2	1	0
ASSR	-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB
Lê/Escrive	L	L/E	L/E	L	L	L	L	L
Valor Inicial	0	0	0	0	0	0	0	0

Bit 6 – EXCLK – Enable External Clock Input

A escrita de 1 neste bit em conjunto com a seleção do *clock* assíncrono permite que um *clock* externo seja a entrada para o pino TOSC1, ao invés do uso de um cristal de 32,768 kHz. O oscilador a cristal só funcionará se este bit estiver em zero.

Bit 5 – AS2 – Asynchronous Timer/Counter 2

A escrita de 1 neste bit ativa o uso do cristal externo de 32,768 kHz para o uso exclusivo do TC2, nos pinos TOSC1 e TOSC2.

Bit 4:0 – TCN2UB, OCR2AUB, OCR2BUB, TCR2AUB e TCR2BUB.

São bits sinalizadores da escrita nos registradores TCNT2, OCR2B, OCR2A, TCCR2A e TCCR2B, respectivamente.

9.4.2 CÓDIGO EXEMPLO

CONTANDO 1 s

O código a seguir é utilizado para gerar um evento periódico de interrupção a cada 1 s. Neste caso, um LED ligado ao pino PB5 troca de estado (liga ou desliga) a cada interrupção. Dentro da rotina de interrupção está o código para a geração de um relógio com a contagem de segundos, minutos e horas. É bom lembrar que quando um registrador de configuração não aparece no código, ou seja, não é escrito, assume-se o seu valor *default*, que geralmente é zero.

TC2_relogio.c

```
//=====================================================================
//          TC2 COM CRISTAL EXTERNO DE 32,768 kHz
//          Rotina para implementar um relógio e piscar um LED
//=====
#define F_CPU 1000000UL      //frequência de operação de 1 MHz
#include <avr/io.h>
#include <avr/interrupt.h>

#define cpl_bit(Y,bit_x) (Y^=(1<<bit_x)) //complementa bit
#define LED PB5

volatile unsigned char segundos, minutos, horas; /*variáveis com os valores para o
                                                 relógio precisam ser tratadas em um código adequado. Variáveis
                                                 globais que são utilizadas dentro de interrupções e lidas fora
                                                 delas devem ser declaradas como volatile. Isto é uma exigência
                                                 para o compilador. Respostas imprevisíveis correrão se o comando
                                                 volatile não for empregado.*/
//-----
ISR(TIMER2_OVF_vect)      //entrada aqui a cada 1 segundo
{
    //rotina para contagem das horas, minutos e segundos
    cpl_bit(PORTB,LED); //piscá LED

    segundos++;

    if(segundos == 60)
    {
        segundos = 0;
        minutos++;

        if(minutos == 60)
        {
            minutos = 0;
            horas++;

            if(horas == 24)
                horas = 0;
        }
    }
}
//-----
int main()
{
    DDRB = 0b00100000;//cristal ext., não importa a config. dos pinos TOSC1 e TOSC2
    PORTB= 0b11011111;//pull-ups habilitados nos pinos não utilizados

    ASSR = 1<<AS2;//habilita o cristal externo para o contador de tempo real
    TCCR2B = (1<<CS2)|(1<<CS20);/*prescaler = 128, freq. p/ o contador = 32.768/128,
                                    o que resulta em uma freq. de 256 Hz. Como o contador
                                    é de 8 bits, ele conta 256 vezes, resultando em um
                                    estouro preciso a cada 1 segundo*/
    TIMSK2 = 1<<TOIE2; //habilita interrupção do TC2

    sei();           //habilita interrupção global

    while(1)
    {}             //código principal (display, ajuste de hora, minutos, etc..)
}
//=====================================================================
```

Exercícios:

9.6 – Qual o valor do divisor de frequência para que o TC2, trabalhando com um cristal externo de 32,768 kHz, produza um estouro a cada 1/4 de segundo?

9.7 – O uso de um cristal 32,768 kHz em relógios digitais tem um motivo óbvio, qual?

9.8 – Elaborar um programa para que o hardware da fig. 9.8 funcione como um relógio 24 h. O ajuste do horário deve ser feito nos botões específicos.

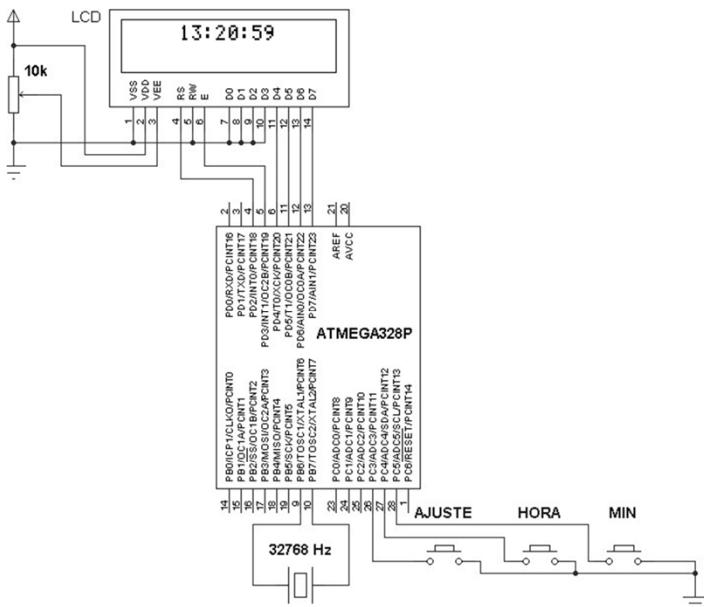


Fig. 9.8 – Relógio com contagem baseado em cristal externo.

9.5 TEMPORIZADOR/CONTADOR 1

O TC1 é um contador de 16 bits que permite grande precisão na geração de formas de onda, temporizações e medição de largura de pulsos. Interrupções podem ser habilitadas para os eventos de estouro ou igualdade de comparação.

A contagem é feita no par de registradores TCNT1H e TCNT1L (contador TCNT1). O TC1 é incrementado via *clock* interno, com ou sem *prescaler*, ou por *clock* externo, ligado ao pino T1. Possuindo dois registradores de controle: TCCR1A e TCCR1B. Os registradores de comparação de saída OCR1A e OCR1B são constantemente comparados com o valor de contagem. O resultado da comparação pode ser usado para gerar sinais PWMs ou com frequência variável nos pinos de saída de comparação (OC1A e OC1B). O valor máximo de contagem (TOP) para o TC1 pode ser definido em alguns modos de operação por OCR1A, ICR1 ou por um conjunto fixo de valores. O registrador de captura de entrada pode capturar o valor do contador quando ocorrer um evento externo no pino ICP1 ou nos pinos do comparador analógico.

MODO NORMAL

É o modo mais simples de operação do contador. Neste modo, a contagem é crescente e contínua, o contador conta de 0 até 65535 (0xFFFF) e volta a zero, sinalizando um estouro e solicitando um pedido de interrupção (se habilitada). O registrador de contagem TCNT1 pode ser escrito ou lido a qualquer momento pelo programa e tem prioridade sobre a contagem ou a limpeza do contador.

MODO DE CAPTURA DE ENTRADA

O modo captura de entrada permite medir o período de um sinal digital externo. Para isso, deve-se ler o valor do TC1 quando ocorrer a interrupção; após duas interrupções, é possível calcular o tempo decorrido entre elas. O

programa deve lidar com os estouros do TC1, caso ocorram. No modo de captura de entrada, escolhe-se qual o tipo de borda do sinal gerará a interrupção (descida ou subida). Para medir o tempo em que um sinal permanece em nível alto ou baixo, após cada captura, é preciso alterar o tipo de borda para a captura.

MODO CTC

No modo CTC (*Clear Timer on Compare*) os registradores OCR1A ou ICR1 são utilizados para mudar a resolução do contador (valor de topo). Neste modo, o contador é limpo (zerado) quando o valor do TCNT1 é igual a OCR1A ou igual a ICR1. Este modo permite grande controle na comparação para estabelecer a frequência de saída e também simplifica a operação de contagem de eventos externos. Um exemplo de diagrama temporal para o modo CTC é apresentado na fig. 9.9; os pinos OC1A e OC1B foram habilitados.

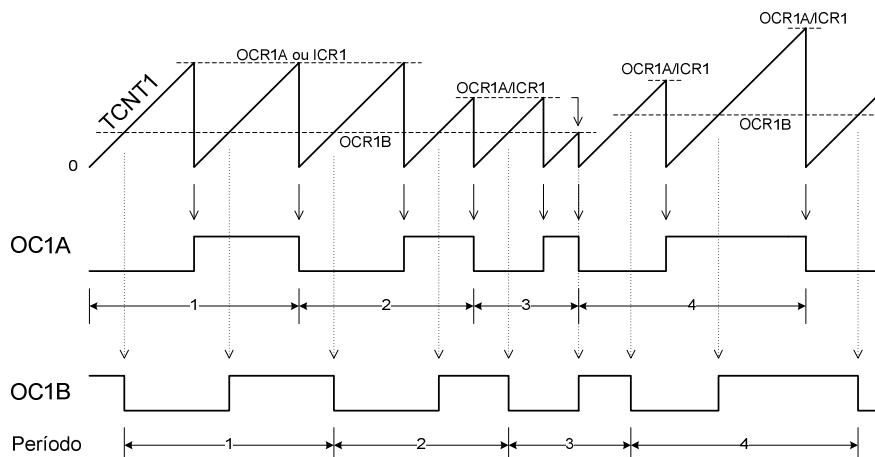


Fig. 9.9 – Diagrama de tempo para o modo CTC.

Para gerar uma onda de saída no modo CTC, os pinos OC1A e OC1B devem ser ajustados para trocar de nível a cada igualdade de comparação. Os valores de OC1A e OC1B não serão visíveis nos seus respectivos pinos, a

não ser que estejam configurados como saída. A máxima frequência que a forma de onda pode alcançar é a metade da frequência de trabalho da CPU. Essa frequência é definida por:

$$f_{OC1A} = \frac{f_{osc}}{2N(1+TOP)} \quad [\text{Hz}] \quad (9.7)$$

onde f_{osc} é a frequência de operação do microcontrolador, TOP tem um valor entre 0 e 65535, e N é o fator do *prescaler* (1, 8, 64, 256 ou 1024). Para calcular o valor TOP em função da frequência desejada, basta isolá-lo na equação acima, o que resulta em:

$$TOP = \frac{f_{osc}}{2N.f_{OC1A}} - 1 \quad (9.8)$$

MODO PWM RÁPIDO

O modo PWM rápido permite a geração de um sinal PWM de alta frequência. O contador conta de zero até o valor máximo e volta a zero. No modo de comparação com saída não-invertida, o pino OC1A é zerado na igualdade entre TCNT1 e OCR1A e colocado em 1 no valor mínimo do contador. No modo de comparação com saída invertida, o processo é inverso: OC1A é ativo na igualdade e zerado no valor mínimo. A comparação para o pino OC1B é feita com OCR1B. O valor de contagem para o TC1 pode ser definido pelos valores fixos 0x0FF (255), 0x1FF (511) ou 0x3FF (1023), por OCR1A ou ICR1; caso OCR1A seja empregado, o pino OC1A não pode gerar uma sinal PWM, apenas uma onda quadrada.

A resolução para o PWM rápido pode ser fixada em 8, 9, 10 bits (0x0FF, 0x1FF ou 0x3FF) ou definida pelos registradores OCR1A ou ICR1. A resolução mínima é de 2 bits (OCR1A ou ICR1 = 3) e a máxima é de 16 bits (ICR1 ou OCR1A = 0xFFFF). A resolução do PWM é calculada com:

$$R_{PWM_Rápido} = \frac{\log(TOP + 1)}{\log(2)} \quad [\text{bits}] \quad (9.9)$$

Neste modo, o contador é incrementado até encontrar um dos valores fixos 0x0FF, 0x1FF ou 0x3FF, o valor de OCR1A ou o valor de ICR1. Um diagrama exemplo para o PWM rápido com OC1A e OC1B como saídas não invertidas é apresentado na fig. 9.10.

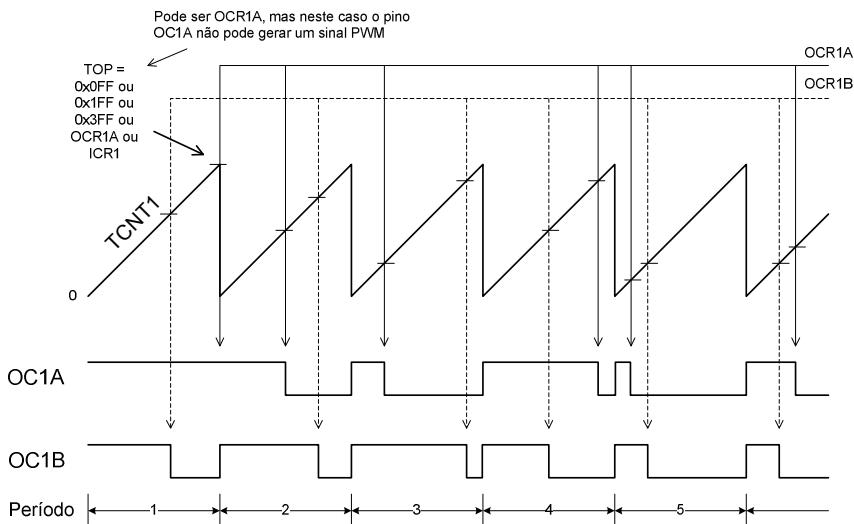


Fig. 9.10 – Diagrama temporal para o modo PWM rápido.

A frequência de saída do PWM rápido é calculada com:

$$f_{OC1x_PWM} = \frac{f_{osc}}{N \cdot (1 + TOP)} \quad [\text{Hz}] \quad (9.10)$$

onde f_{osc} é a frequência de operação do microcontrolador e N é o fator do prescaler (1, 8, 64, 256 ou 1024). Um sinal com ciclo ativo de 50% pode ser obtido ajustando-se OC1A para mudar de nível lógico a cada igualdade de comparação (onda quadrada). Isso se aplica apenas quando OCR1A é usado para definir o valor TOP. A frequência máxima gerada será a metade da frequência de trabalho da CPU quando OCR1A = 0. Quando se controla o ciclo ativo do sinal PWM, o valor zero para OCR1A produz um pequeno ruído, e o valor máximo (TOP) vai deixar o sinal PWM em 0 ou 1, conforme foi habilitada a saída, invertida ou não.

MODO PWM COM FASE CORRIGIDA

O modo PWM com fase corrigida permite ajustar a fase do PWM, isto é, o início e fim do ciclo ativo do sinal PWM. É baseado na contagem crescente e decrescente do TCNT1, que conta repetidamente de zero até o valor máximo (TOP) e vice-versa (permanece um ciclo de *clock* no valor TOP), o que torna a saída PWM simétrica dentro de um período do PWM. Além disso, o valor de comparação que determina o razão cíclica da saída PWM é armazenado em registradores, sendo atualizado apenas quando o contador atinge o valor TOP, o qual marca o início e o fim de um ciclo PWM. Assim, se o valor TOP não for alterado com o temporizador em operação, o pulso gerado será sempre simétrico em relação ao ponto médio do período (TCNT1 = 0x00), qualquer que seja a razão cíclica.

Para o sinal PWM não invertido, o pino OC1A é zerado na igualdade entre TCNT1 e OCR1A, quando a contagem é crescente, e colocado em 1 quando a contagem é decrescente. No modo de comparação com saída invertida, o processo é o contrário. A comparação para o pino OC1B é feita com o registrador OCR1B. O valor máximo de contagem pode ser definido como 0x0FF, 0x1FF, 0x3FF, pelo valor de OCR1A ou por ICR1, como no modo PWM rápido. Se OCR1A for utilizado para determinar o valor TOP, o pino OC1A não poderá gerar um sinal PWM, somente uma onda quadrada. A resolução do PWM é dada pela mesma equação do modo PWM rápido (eq. 9.9). Um diagrama de tempo exemplo, para o PWM com correção de fase, é mostrado na fig. 9.11 para OC1A como saída invertida e OC1B como saída não-invertida.

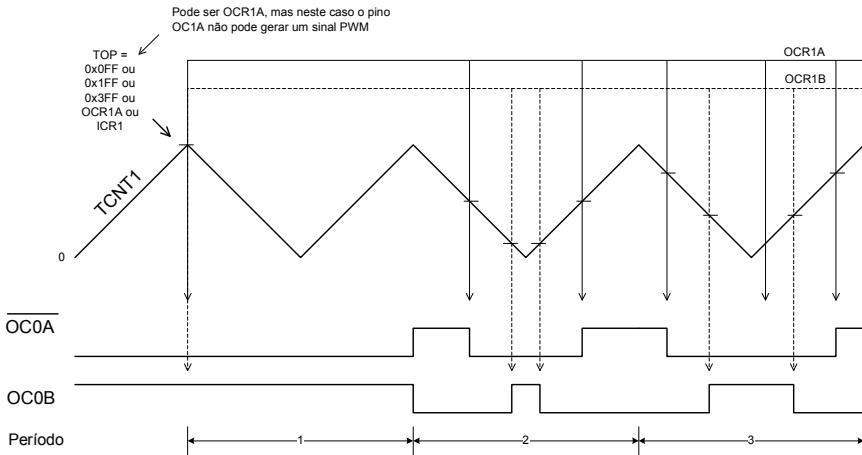


Fig. 9.11 – Diagrama de tempo para o PWM com fase corrigida, saída OC1A invertida e OC1B não invertida.

A frequência de saída do PWM com fase corrigida é calculada com:

$$f_{OC1A_PWM} = \frac{f_{osc}}{2N \cdot TOP} \quad [\text{Hz}] \quad (9.11)$$

onde f_{osc} é a frequência de operação do microcontrolador e N é o fator do prescaler (1, 8, 64, 256 ou 1024). Se OCR1A definir o valor TOP, a saída OC1A terá um ciclo ativo de 50%.

MODO PWM COM FASE E FREQUÊNCIA CORRIGIDAS

Este modo é igual ao PWM com fase corrigida, a diferença principal é que o pulso gerado será sempre simétrico em relação ao ponto médio do período (neste modo PWM, o TOP), mesmo quando o valor de TOP é alterado com o TC1 em operação. Isso ocorre porque os registradores de comparação são atualizados quando TCNT1 = 0x00, permitindo que o momento da comparação nas inclinações de subida e de descida seja sempre o mesmo (para mais detalhes consultar o manual do ATmega328).

9.5.1 REGISTRADORES DO TC1

O controle do modo de operação do TC1 é feito nos registradores TCCR1A e TCCR1B (*Timer/Counter 1 Control Registers*).

Bit	7	6	5	4	3	2	1	0
TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10
Lê/Escr. Valor Inicial	L/E 0	L/E 0	L/E 0	L/E 0	L 0	L 0	L/E 0	L/E 0

Bits 7:6 – COM1A1:0 – Compare Output Mode for channel A

Bits 5:4 – COM1B1:0 – Compare Output Mode for channel B

COM1A1:0 e COM1B1:0 controlam o comportamento dos pinos OC1A e OC1B, respectivamente. Se o valor 1 for escrito nesses bits a funcionalidade dos pinos é alterada (os bits no registrador DDRx correspondentes a OC1A e OC1B devem ser colocados em 1 para habilitar cada *driver* de saída). Quando OC1A e OC1B forem empregados, a funcionalidade dos bits COM1x1:0 dependerá do ajuste dos bits WGM13:0. Nas tabs. 9.17-19, são apresentadas as configurações dos bits COM1x1:0 para os diferentes modos de operação do TC1.

Tab. 9.17– Modo não PWM (normal e CTC).

COM1A1/COM1B1	COM1A0/COM1B0	Descrição
0	0	Operação normal dos pinos, OC1A/OC1B desconectados.
0	1	Mudança de OC1A/OC1B na igualdade de comparação.
1	0	Limpeza de OC1A/OC1B na igualdade de comparação (saída em nível lógico baixo).
1	1	OC1A/OC1B ativos na igualdade de comparação (saída em nível lógico alto).

Tab. 9.18 – Modo PWM rápido.

COM1A1/COM1B1	COM1A0/COM1B0	Descrição
0	0	Operação normal dos pinos, OC1A/OC1B desconectados.
0	1	WGM13:0 = 14 ou 15: Mudança de OC1A na igualdade de comparação, OC1B desconectado (operação normal do pino). Para os demais valores de WGM1, OC1A/OC1B estarão desconectados (operação normal dos pinos).
1	0	Limpeza de OC1A/OC1B na igualdade de comparação, ativos no valor mínimo de comparação (modo não invertido).
1	1	OC1A/OC1B ativos na igualdade de comparação, limpos no valor mínimo de comparação (modo invertido).

Tab. 9.19 – Modo PWM com correção de fase e correção de fase e frequência.

COM1A1/COM1B1	COM1A0/COM1B0	Descrição
0	0	Operação normal dos pinos, OC1A/OC1B desconectados.
0	1	WGM13:0 = 9 ou 11: Mudança de OC1A na igualdade de comparação, OC1B desconectado (operação normal do pino). Para os demais valores de WGM1, OC1A/OC1B estarão desconectados (operação normal dos pinos).
1	0	Limpeza de OC1A/OC1B na igualdade de comparação quando a contagem é crescente, ativos no valor mínimo de comparação quando a contagem é decrescente.
1	1	OC1A/OC1B ativos na igualdade de comparação quando a contagem é crescente, limpos no valor mínimo de comparação quando a contagem é decrescente.

Bits 1:0 – WGM11:0 – Waveform Generation Mode

Combinados com os bits WGM13:2 do registrador TCCR1B, esses bits controlam a forma de contagem do contador, a fonte para o valor máximo (TOP) e qual tipo de forma de onda gerada será empregada (tab. 9.20).

Tab. 9.20 – Descrição dos bits para os modos de geração de formas de onda.

Modo	WGM 13	WGM 12	WGM 11	WGM 10	Modo de operação do TC1	Valor TOP	Atualiz. OCR1x no valor	Bit TOV1 ativo no valor:
0	0	0	0	0	Normal	0xFFFF	Imediata	0xFFFF
1	0	0	0	1	PWM com fase corrigida, 8 bits	0x00FF	0x00FF	0
2	0	0	1	0	PWM com fase corrigida, 9 bits	0x01FF	0x01FF	0
3	0	0	1	1	PWM com fase corrigida, 10 bits	0x03FF	0x03FF	0
4	0	1	0	0	CTC	OCR1A	Imediata	0xFFFF
5	0	1	0	1	PWM rápido, 8 bits	0x00FF	0	0x00FF
6	0	1	1	0	PWM rápido, 9 bits	0x01FF	0	0x01FF
7	0	1	1	1	PWM rápido, 10 bits	0x03FF	0	0x03FF
8	1	0	0	0	PWM com fase e freq. corrigidas	ICR1	0	0
9	1	0	0	1	PWM com fase e freq. corrigidas	OCR1A	0	0
10	1	0	1	0	PWM com fase corrigida	ICR1	ICR1	0
11	1	0	1	1	PWM com fase corrigida	OCR1A	OCR1A	0
12	1	1	0	0	CTC	ICR1	Imediata	0xFFFF
13	1	1	0	1	Reservado	-	-	-
14	1	1	1	0	PWM rápido	ICR1	0	ICR1
15	1	1	1	1	PWM rápido	OCR1A	0	OCR1A

TCCR1B - Timer/Counter 1 Control Register B

Bit	7	6	5	4	3	2	1	0
TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
Lê/Escr.	L/E	L/E	L	L/E	L/E	L/E	L/E	L/E
Valor	0	0	0	0	0	0	0	0
Inicial								

Bit 7 – ICNC1 – Input Capture Noise Canceler

Colocando este bit em 1, o filtro de ruído do pino de captura ICP1 é habilitado. Esse filtro requer 4 amostras sucessivas iguais para o ICP1 mudar sua saída. Assim, a captura de entrada é atrasada por 4 ciclos de *clock*.

Bit 6 – ICES1 – Input Capture Edge Select

Este bit seleciona qual borda no pino de entrada de captura (ICP1) será usada para disparar o evento de captura (ICES1=0 na transição de 1 para 0, ICES1=1 na transição de 0 para 1). Quando uma captura ocorre, o valor do contador é copiado no registrador ICR1.

Bit 4:3 – WGM13:2 – Waveform Generation Mode

Ver a tab. 9.20.

Bit 2:0 – CS12:0 – Clock Select

Existem 3 bits para a escolha da fonte de *clock* para o TC1 (tab. 9.21).

Tab. 9.21 – Descrição dos bits para seleção do *clock* para o TC1.

CS12	CS11	CS10	Descrição
0	0	0	Sem fonte de <i>clock</i> (TC1 parado).
0	0	1	<i>clock</i> /1 (prescaler = 1) sem prescaler.
0	1	0	<i>clock</i> /8 (prescaler = 8) .
0	1	1	<i>clock</i> /64 (prescaler = 64).
1	0	0	<i>clock</i> /256 (prescaler = 256).
1	0	1	<i>clock</i> /1024(prescaler = 1024).
1	1	0	Fonte de <i>clock</i> externa no pino T1 (contagem na borda de descida).
1	1	1	Fonte de <i>clock</i> externa no pino T1 (contagem na borda de subida).

TCCR1C - Timer/Counter 1 Control Register C

Bit	7	6	5	4	3	2	1	0
TCCR1C	FOC1A	FOC1B	-	-	-	-	-	-
Lê/Escr.	L/E	L/E	L	L	L	L	L	L
Valor	0	0	0	0	0	0	0	0
Inicial								

Bits 7:6 – FOC2A:B – Force Output Compare A e B

Estes bits são ativos somente para os modos não-PWM. Quando em 1, obrigam uma comparação no módulo gerador de forma de onda. O efeito nas saídas dependerá da configuração dado aos bits COM1A1:0 e COM1B1:0.

TCNTH e TCNTL (TCNT1) – Timer/Counter 1 Register

São os dois registrador de 8 bits onde é realizada a contagem do TC1, H (*high*) L (*Low*), podem ser lidos ou escritos a qualquer tempo.

OCR1AH e OCR1AL (OCR1A) – Output Compare 1 Register A

Registradores de comparação A de 8 bits cada, possui o valor que é continuamente comparado com o valor do contador (TCNT1). A igualdade pode ser utilizada para gerar uma interrupção ou uma forma de onda no pino OC1A.

OCR1BH e OCR1BL (OCR1B) – Output Compare 1 Register B

Registradores de comparação B de 8 bits cada, possui o valor que é continuamente comparado com o valor do contador (TCNT1). A igualdade pode ser utilizada para gerar uma interrupção ou uma forma de onda no pino OC1B.

ICR1H e ICR1L (ICR1) – Input Capture Register 1

Esses registradores são atualizados com o valor do TCNT1 cada vez que um evento ocorre no pino ICP1 (ou opcionalmente nos pinos do comparador analógico). Também são empregados para definir o valor máximo de contagem (TOP).

TIMSK1 – Timer/Counter 1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
TIMSK1	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1
Lê/Escrive	L	L	L/E	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 5 – ICIE1 – Timer/Counter 1, Input Capture Interrupt Enable

A escrita de 1 neste bit ativa a interrupção por captura da entrada. Quando ocorre uma mudança no pino ICP1 ou nos pinos do comparador analógico, o valor do TCNT1 é salvo no registrador ICR1.

Bit 2 – OCIE1B – Timer/Counter 1 Output Compare Match B Interrupt Enable

A escrita de 1 neste bit ativa a interrupção do TC1 na igualdade de comparação com o registrador OCR1B.

Bit 1 – OCIE1A – Timer/Counter 1 Output Compare Match A Interrupt Enable

A escrita de 1 neste bit ativa a interrupção do TC1 na igualdade de comparação com o registrador OCR1A.

Bit 0 – TOIE1 – Timer/Counter 1 Overflow Interrupt Enable

A escrita de 1 neste bit ativa a interrupção por estouro do TC1.

As interrupções individuais dependem da habilitação das interrupções globais pelo bit I do registrador SREG.

TIFR1 – Timer/Counter 1 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
TIFR2	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1
Lê/Escrive	L	L	L/E	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 5 – ICF1 – Timer/Counter 1 Input Capture Flag

Este bit é colocado em 1 quando um evento relacionado ao pino ICP1 ocorre.

Bit 2 – OCF1B – Timer/Counter 1 Output Compare B Match Flag

Este bit é colocado em 1 quando o valor da contagem (TCNT1) é igual ao valor do registrador de comparação de saída B (OCR1B) do TC1.

Bit 1 – OCF1A – Timer/Counter 1 Output Compare A Match Flag

Este bit é colocado em 1 quando o valor da contagem (TCNT1) é igual ao valor do registrador de comparação de saída A (OCR1A) do TC1.

Bit 0 – TOV1 – Timer/Counter 1 Overflow Flag

Este bit é colocado em 1 quando um estouro do TC1 ocorre.

9.5.2 CÓDIGOS EXEMPLO

MODO NORMAL – GERANDO UM PEDIDO DE INTERRUPÇÃO

O código abaixo é utilizado para gerar um evento periódico de interrupção a cada 4,19 s. Neste caso, um LED ligado ao pino PB5 troca de estado (liga ou desliga) a cada interrupção. Foi empregado o maior divisor de *clock* possível, 1024. Em relação ao exemplo apresentado na seção 9.3.2 (TC0), percebe-se o aumento considerável no tempo de temporização, pois, desta vez, o TC1 conta 65536 vezes ao invés de apenas 256.

TC1_estouro.c

```
//==============================================================================
// HABILITANDO A INTERRUPÇÃO POR ESTOURO DO TC1
//==============================================================================
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#define cpl_bit(y,bit) (y^=(1<<bit))//troca o estado lógico do bit x da variável Y
#define LED    PB5
//-----
ISR(TIMER1_OVF_vect)           //interrupção do TC1
{
    cpl_bit(PORTB,LED);
}
//-----
```

```

int main()
{
    DDRB = 0b00100000; //somente pino do LED como saída
    PORTB = 0b11011111; //apaga LED e habilita pull-ups nos pinos não utilizados

    TCCR1B = (1<<CS12) | (1<<CS10); /*TC1 com prescaler de 1024, a 16 MHz gera uma
                                         interrupção a cada 4,19 s*/
    TIMSK1 = 1<<TOIE1; //habilita a interrupção do TC1
    sei(); //habilita interrupções globais

    while(1)
    {
        /*Aqui vai o código, a cada estouro do TC1 o programa desvia para ISR(TIMER1_OVF_vect)*/
    }
}
//=====

```

MODO DE CAPTURA DE ENTRADA

O programa abaixo utiliza um botão para gerar uma interrupção por captura no pino ICP1. Não foi feito o tratamento do ruído do botão e cada vez que o botão é pressionado o pino PB5 troca de estado. O valor do registrador ICR1 é utilizado para ler o valor do TC1.

TC1_captura.c

```

//=====
//      HABILITANDO A INTERRUPÇÃO POR CAPTURA          //
//=====

#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#define cpl_bit(y,bit) (y^(1<<bit))//troca o estado lógico do bit x da variável Y
#define LED          PBS
//-----
/*interrupção do TC1, toda vez que ocorrer um evento no pino ICP1 (PB0) ICR1 terá o
valor de contagem do TCNT1*/
//-----
ISR(TIMER1_CAPT_vect)
{
    cpl_bit(PORTB,LED); //troca estado do pino PB5
}
//-----
int main()
{
    DDRB = 0b00100000; //somente pino do LED como saída, botão no PB0
    PORTB = 0b11011111; //apaga LED e habilita pull-ups nos outros pinos
    TCCR1B = 1<<CS10; //TC1 com prescaler = 1, captura na borda de descida
    TIMSK1 = 1<<ICIE1; //habilita a interrupção por captura
    sei(); //habilita interrupções globais

    while(1)
    {
        /*Aqui vai o código, a cada evento no pino ICP1 o programa desvia para
         ISR(TIMER1_CAPT_vect)*/
    }
}
//=====

```

MODO CTC

TC1_PWMs.c

```
#define F_CPU 16000000UL
#include <avr/io.h>

int main(void)
{
    DDRB = 0b00000110; //pinos OC1B e OC1A (PB2 e PB1) como saída
    PORTB = 0b11111001; //zera saídas e habilita pull-ups nos pinos não utilizados

    //MODO CTC - TOP = ICR1
    TCCR1A = 0b01010000; /*habilita OC1A e OC1B para trocar de estado na igualdade de
                           comparação*/
    TCCR1B = 0b00011011; //liga TC1 com prescaler = 64.
    ICR1 = 10000;         //valor máximo de contagem

    while(1)
    {}                  //o programa principal vai aqui
}
```

MODO PWM RÁPIDO

```
...
//fast PWM, TOP = ICR1, OC1A e OC1B habilitados
TCCR1A = 0b10100010;      //PWM não invertido nos pinos OC1A e OC1B
TCCR1B = 0b00011001;      //liga TC1, prescaler = 1
ICR1 = 35000;             //valor máximo para contagem
OCR1A = 2000;             //controle do ciclo ativo do PWM OC1A
OCR1B = 100;              //controle do ciclo ativo do PWM OC1B
...

...
```

MODO PWM COM FASE E FREQUÊNCIA CORRIGIDAS

```
...
//phase and frequency correct PWM, TOP = OCR1A
TCCR1A = 0b00100011;      // OC1B habilitado, modo não invertido
TCCR1B = 0b00011001;      //liga TC1, prescaler = 1 e habilita
OCR1A = 300;               //máximo valor para contagem
OCR0B = 100;              //controle do ciclo ativo do PWM OC1B
...

...
```

MODO CTC – CONTANDO EVENTOS EXTERNOS

Abaixo, é apresentado um programa para gerar uma interrupção a cada 1 s baseado num sinal de 60 Hz externo ao microcontrolador. Esse tipo de sinal, por exemplo, é utilizado em rádios-relógio para gerar a base

de tempo. Na fig. 9.12, é apresentado o circuito para o programa. O sinal de 60 Hz é digital, sendo, usualmente obtido da rede elétrica após um tratamento adequado.

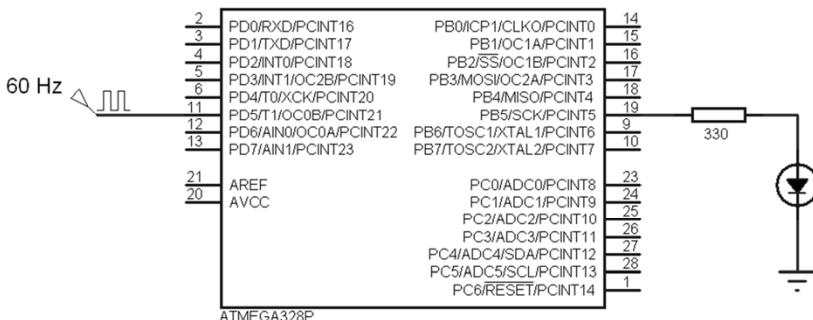
TC1_externo.c

```
//=====
//      TC1 estouro na igualdade de comparação - sinal externo          //
//      Pisca LED a cada 1 segundo                                     //
//=====
#include <avr/io.h>
#include <avr/interrupt.h>

#define cpl_bit(Y,bit_X) (Y^=(1<<bit_X))
#define LED          PB5
//-----
ISR(TIMER1_COMPA_vect)    //sub-rotina de interrupção por igualdade de comparação
{
    cpl_bit(PORTB,PB5); //troca o estado do LED do pino PB5
}
//-----
int main()
{
    DDRD = 0x00;           //PORTD será a entrada do sinal de clock para o TC1 (PD3)
    DDRB = 1<<PB5;        //pino PB5 é a saída para o LED de sinalização

    TIMSK1 = 1<<OCIE1A; //habilita a interrupção do TC1 por igualdade de comparação
    TCCR1B = (1<<WGM12)|(1<<CS12) | (1<<CS11) |(1<<CS10); /*clock externo contagem na
                           borda de subida - modo CTC*/
    OCR1A = 59;           /*valor para a contagem máxima do TC1 (conta 60 vezes) - valor de
                           comparação. Como o sinal de clock externo é de 60 Hz, é gerada
                           uma interrupção a cada 1 s*/
    sei();                //liga a interrupção

    while(1){}
}
//=====
```



Exercícios:

9.9 – Considerando o ATmega328 rodando a 16MHz e o TC1 operando no modo normal (65536 contagens), quanto tempo levará para ocorrer um estouro do TC1 para cada *prescaler* possível?

9.10 – Elaborar um programa para que o ATmega328 funcione como frequencímetro digital, apresentando o valor lido em um LCD 16×2. Empregue um circuito adequado (sugestão na fig. 9.13).

Sugestão: utilize um TC de 8 bits para gerar uma base de tempo de 1 s e o TC1, de 16 bits, para contar os eventos externos. A frequência será dada pelo número de contagem do TC de 16 bits (se houver estouro, o programa deve levar esse fato em conta no cômputo da frequência).

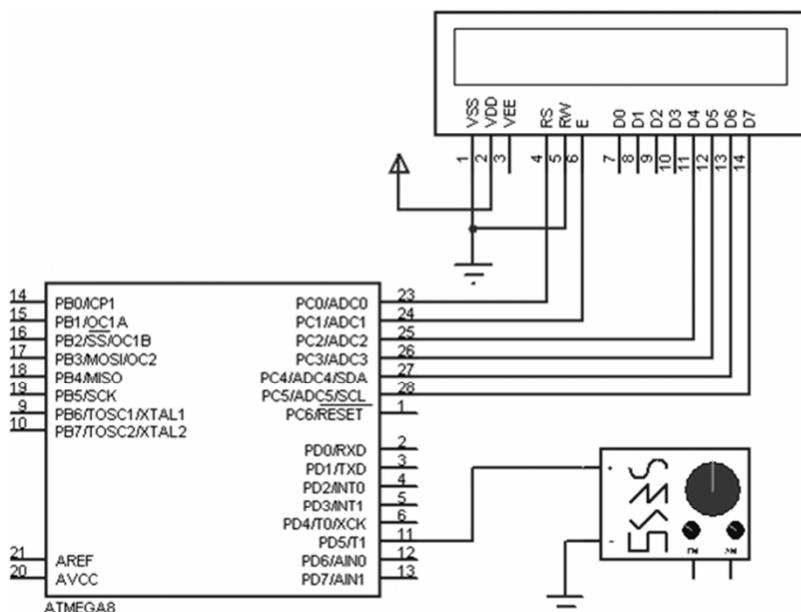


Fig. 9.13 – Circuito para o exercício 9.10 (obs.: a pinagem do ATmega8 é igual a do ATmega328).

9.6 PWM POR SOFTWARE

Algumas vezes pode ser necessário criar sinais PWM unicamente através da programação. Isso ocorre quando o microcontrolador não dispõe do número desejado de canais PWM, ou os mesmos não apresentam a resolução necessária. No ATmega328 estão disponíveis 6 sinais PWMs, dos quais, 4 são de 8 bits e 2 de 16 bits. Porém, os pinos que podem gerar esses sinais são estáticos (não podem ser alterados), não permitindo muita flexibilidade na hora de projetar o hardware.

Se for inviável substituir o microcontrolador por um que satisfaça os requisitos de projeto, o problema deve ser resolvido por programação. A ideia é simples: utilizar um temporizador para gerar a base de tempo para o sinal PWM, cuja resolução ficará dependente das variáveis utilizadas e do número de contagens. O programa a seguir ilustra essa ideia.

PWM_software.c

```
//===================================================================== //
//      PWM borda simples - não invertido, via software           //
//===================================================================== //
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>

#define    set_bit(y,bit)  (y|=(1<<bit)) //coloca em 1 o bit x da variável Y
#define    clr_bit(y,bit)  (y&=~(1<<bit)) //coloca em 0 o bit x da variável Y

#define PWM1          PB0      //escolha do pino para o sinal PWM1
#define Resol_PWM1   1000     //PWM1 com 1000 passos temporais de resolução

volatile unsigned int Passo_PWM1 = 0;
volatile unsigned int Ciclo_Ativo_PWM1;

//-----
ISR(TIMER0_OVF_vect)//o tempo de estouro do TCO determina a menor resolução temporal
{
    //para o PWM (ciclo ativo)
    Passo_PWM1++; //incremento do passo de tempo

    if(Passo_PWM1==Resol_PWM1)
    {
        Passo_PWM1=0;      //inicializa o contador
        set_bit(PORTB,PWM1); //na igualdade de comparação coloca o pino PWM1 em 1
    }

    if(Passo_PWM1==Ciclo_Ativo_PWM1)
        clr_bit(PORTB,PWM1); /*quando o contador atinge o valor do ciclo ativo do
                                PWM1 o pino vai a zero*/
}
//-----
```

```

int main()
{
    DDRB = 0b00000001;//somente o pino do LED como saída
    PORTB = 0b11111110;//apaga o LED e habilita pull-ups nos pinos não utilizados

    TCCR0B = 1<<CS00;//TC0 com prescaler de 1, a 16 MHz gera uma interrupção a cada 16 us
    TIMSK0 = 1<<TOIE0;//habilita a interrupção do TC0
    sei();           //habilita interrupções globais

    Ciclo_Ativo_PWM1 = 500;//determinação do ciclo ativo para o PWM1

    while(1)
    {}           //Aqui vai o código principal
}

//-----

```

O programa acima utilizou uma resolução de 1000 pontos (Resol_PWM1) e o TC0 foi configurado para trabalhar sem divisão de *clock*. Desta forma, o tempo de estouro para um frequência de trabalho de 16 MHz foi de 16 µs (256/16MHz). Isso significa que a cada 16 us é gerada uma interrupção, na qual é executada a rotina para gerar o sinal PWM. Considerando o número de pontos do sinal PWM e a menor resolução temporal possível para o ciclo ativo, resulta em um período de 16 ms para o sinal, ou seja, uma frequência de 62,5 Hz. O ciclo ativo do sinal é controlado por uma variável própria e é escolhido qual pino disponibilizará o sinal PWM.

Para alterar a frequência do sinal PWM é necessário alterar a sua resolução, ou seja, o tempo de estouro do TC0. Lembrando-se que é possível configurar diversos modos de trabalho para os TCs no ATmega.

Exercício

9.11 – Faça um programa para gerar 4 sinais PWM com resolução de 100, 400, 1000 e 5000 pontos. Escolha a forma de trabalho do TC empregado.

Qual é a frequência dos sinais gerados se for empregado uma frequência para o trabalho da CPU de 20 MHz?

9.7 ACIONANDO MOTORES

Nesta seção, são apresentados alguns conceitos básicos de motores comumente utilizados com circuitos eletrônicos de baixa potência, os quais podem ser acionados diretamente por chaves transistorizadas ou *drivers* de corrente de baixa potência. Uma abordagem profunda está longe do escopo deste trabalho. Recomenda-se, assim, que bibliografias especializadas sejam consultadas para o completo entendimento do funcionamento dos referidos motores.

9.7.1 MOTOR DC DE IMÃ PERMANENTE

Os motores DC de imã permanente são muito utilizados em equipamentos eletrônicos, sendo encontrados em aparelhos de CD/DVD, *Blue Ray*, computadores (ventiladores) e em muitos dispositivos que exigem movimento de partes mecânicas sem grande precisão. Para aplicações eletrônicas, geralmente operam com baixas tensões, até uns 12 V, e possuem pequenas dimensões. Alguns tipos de motores DC são apresentados na fig. 9.14.



Fig. 9.14 – Motores DC encontrados em aparelhos de CD/DVD de computadores.

O movimento dos motores DC é baseado na interação entre o campo magnético de seu(s) imã(s) permanente(s) e o campo magnético gerado pela corrente que percorre suas bobinas. Sua rotação é dependente da tensão média aplicada aos seus terminais. Dessa forma, com o emprego de um sinal PWM, é possível controlar a velocidade de um motor DC.

Para a inversão do sentido de rotação do motor é necessário inverter a tensão de alimentação do motor e, logo, sua corrente. Para alcançar tal funcionalidade deve ser empregado um circuito com transistores funcionando como chaves, a chamada ponte H, ilustrada na fig. 9.15. Para o motor girar para a direita devem ser acionados os transistores Q1 e Q2, então a corrente terá o sentido de I_D ; para girar para a esquerda, devem ser acionados os transistores Q3 e Q4, gerando a corrente I_E (maiores detalhes sobre chaves transistorizadas podem ser vistos no apêndice D).

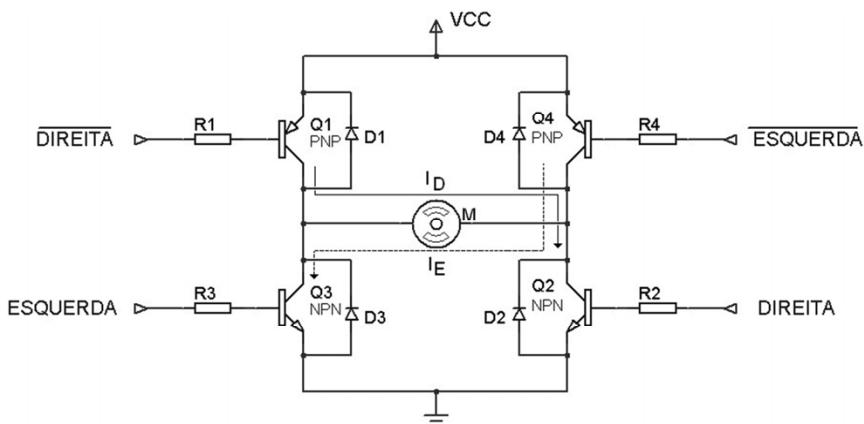


Fig. 9.15 – Ponte H transistorizada para o controle do sentido de rotação de um motor DC.

Um circuito de controle é necessário para gerar adequadamente os sinais para a ponte H, geralmente comandado por um microcontrolador.

A ponte H apresentada na fig. 9.15 é um tanto simplificada, as tensões de controle devem ser adequadas ao valor da tensão de

alimentação do motor. Circuitos adaptadores (*buffers*) podem ser necessários para fazer o casamento entre os sinais digitais de controle e os níveis de tensão da ponte H. Da mesma forma, de acordo com as potências envolvidas, as chaves transistorizadas deverão ser adequadamente projetadas.

Exercícios:

9.12 – Utilizando o circuito da fig. 9.16, elaborar um programa para controlar, através de um sinal PWM, um motor DC com 256 níveis de velocidade. O nível da velocidade selecionado deve ser apresentado no *display* (valor hexadecimal) e armazenado na memória EEPROM para a inicialização do motor. O display de 7 segmentos deve apresentar um número entre 00 e FF.

Obs.: é aconselhado que a alimentação do motor seja independente do circuito de controle, pois podem haver picos de corrente prejudiciais. No caso do microcontrolador, ele pode ser reinicializado. Tudo vai depender das características da fonte de alimentação

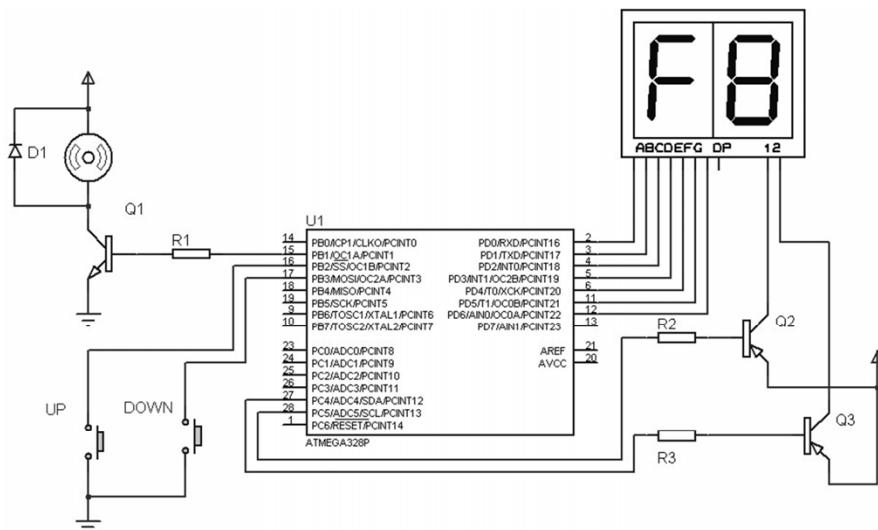


Fig. 9.16 – Controle de um motor DC com um sinal PWM.

9.13 – Existem circuitos integrados específicos para o controle do sentido de rotação de motores DC, tal como o L298. Consulte o manual do fabricante para entender o seu funcionamento. Quais seriam os pinos do L298 onde um sinal PWM poderia ser empregado?

Para o Arduino, existem módulos prontos para o trabalho com motores DC, como o apresentado na fig. 9.17, o qual emprega o L298 e o ULN2803.

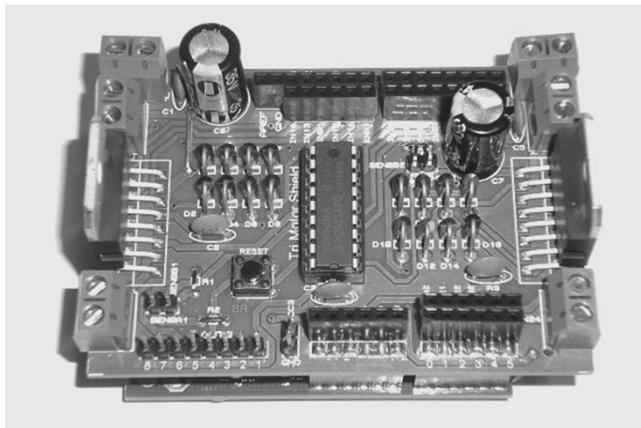


Fig. 9.17 – Módulo para controle de motores para o Arduino²⁹.

9.7.2 MICRO SERVO MOTOR

Os servo motores apresentam movimentação precisa do seu eixo de acordo com ângulos precisos de movimento, sendo adequados para o posicionamento de partes mecânicas. Em eletrônica, são encontrados em aeromodelismo e outros dispositivos mecânicos de controle remoto, sendo chamados de servo motores, servo para robista, ou micro servo, dado suas pequenas dimensões. São constituídos por um motor DC de imãs permanentes, um circuito de controle interno, incluindo os *drivers* necessários para acionar o motor e, ainda, por engrenagens de redução. Essas características os tornam compactos, robustos e fáceis de usar. Na fig. 9.18, é possível ver um micro servo. À direita nessa figura, ele está

²⁹ Circuito esquemático, PCB e demais detalhes podem ser encontrados em www.borgescorporation.blogspot.com

aberto, permitindo a visualização de suas partes constituintes: motor DC, circuito de controle e engrenagens.



Fig. 9.18 – Micro servo motor.

Os servo motores possuem 3 fios: dois de alimentação e um de controle. O interessante é que o ângulo de giro do motor, a saída do sistema de engrenagens, é determinado pela largura do ciclo ativo de um sinal PWM aplicado no controle, como exemplificado na fig. 9.19. O período do sinal é de 20 ms (50 Hz) e o ciclo ativo geralmente pode variar de 0,5 ms até 2,5 ms (vai depender das especificações do fabricante).

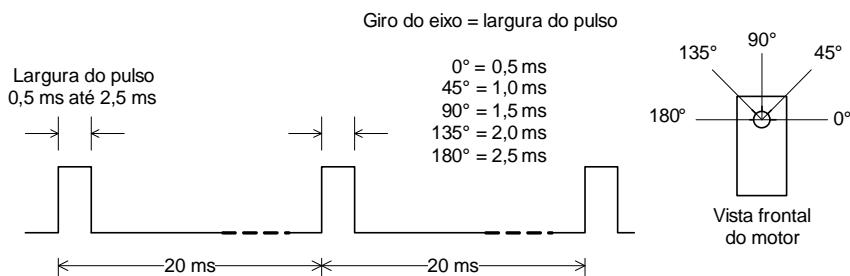


Fig. 9.19 – Sinal de controle para um micro servo motor.

Com base no sinal de controle do servo é possível utilizar o ATmega328 para gerar o sinal PWM adequado, conforme exemplificado pelo programa a seguir, empregando o TC1.

Controle_servo_motor.c

```
/*
----- EXEMPLO DO CONTROLE DE DOIS MOTORES SERVO COM SINAIS PWM
Uso do TC1 no modo PWM rápido, não invertido, pinos OC1A e OC1B
Valor TOP de contagem em ICR1
-----*/
#define F_CPU 16000000UL
#include <avr/io.h>

//Definições de macros
#define set_bit(adress,bit) (adress|=(1<<bit))
#define clr_bit(adress,bit) (adress&=~(1<<bit))

#define TOP 39999 //valor para a máxima contagem

int main()
{
    DDRB = 0b00000110;//habilita os pinos OC1A e OC1B (PB1 e PB2) como saídas
    PORTB = 0b11111001;
                //TOP = (F_CPU/(N*F_PWM))-1, com N = 8 e F_PWM = 50 Hz
    ICR1 = TOP; //configura o período do PWM (20 ms)

    // Configura o TC1 para o modo PWM rápido via ICR1, prescaler = 8
    TCCR1A = (1 << WGM11);
    TCCR1B = (1 << WGM13) | (1<<WGM12) | (1 << CS11);

    set_bit(TCCR1A,COM1A1); //ativa o PWM no OC1B, modo de comparação não-invertido
                           //para desabilitar empregar clr_bit(TCCR1A, COM1A1)
    set_bit(TCCR1A,COM1B1); //ativa o PWM no OC1A, modo de comparação não-invertido
                           //para desabilitar empregar clr_bit(TCCR1A, COM1B1)

    //Pulso de 2 ms em OC1A
    OCR1A = 4000; /*regra de três para determinar este valor: ICR1(TOP) = 20 ms,
                   OCR1A (4000) = 2 ms*/
    //Pulso de 1 ms em OC1B
    OCR1B = 2000; /*regra de três para determinar este valor: ICR1(TOP) = 20 ms,
                   OCR1B (2000) = 1 ms*/
    while(1)
    {}//programa principal
}
//-----
```

Dada as características do servo motor: controle e engrenagens, alguns projetistas gostam de empregá-los para substituir os motores DC em algumas aplicações, como por exemplo, em pequenos robôs. Existem motores servo que não possuem limite de giro (ângulo) e podem ser acionados continuamente. Todavia, é possível alterar os servos comuns

para que executem giros contínuos. A inversão de giro é feita com a alteração da largura do pulso de controle.

Exercício:

9.14 – Elaborar um programa para controlar o ângulo de giro de um micro servo motor de acordo com dois botões, fig. 9.20. Começando com um pulso de 0,5 ms até 2,5 ms, com passo de incremento de 0,1 ms. Repita o procedimento para um passo de 0,05 ms. Qual a relação angular de giro com a largura de pulso?

Obs.: é aconselhado que a alimentação do motor seja independente do circuito de controle, pois podem haver picos de corrente prejudiciais, no caso do microcontrolador, ele pode ser inicializado. Tudo vai depender das características da fonte de alimentação.

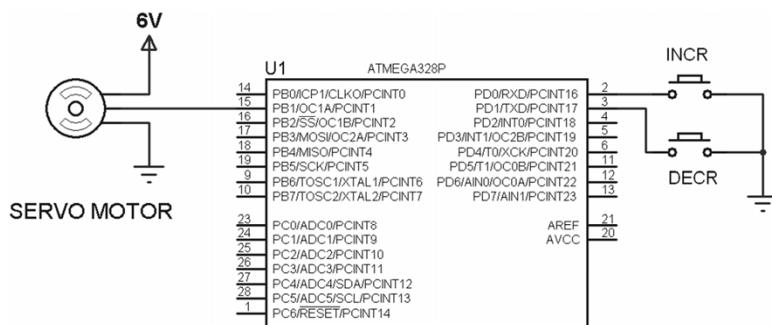


Fig. 9.20 – Circuito para o acionamento de um servo motor.

9.7.3 MOTOR DE PASSO UNIPOLAR

O motor de passo produz uma rotação precisa em seu eixo com passos angulares discretos de acordo com a sequência de energização de suas bobinas. É muito utilizado em sistemas que necessitam de um posicionamento preciso de algum componente mecânico, como por exemplo, impressoras e robôs. A seguir, será descrito o motor de passo do tipo unipolar com rotor de ímã permanente. Na fig.9.21, alguns motores de

passo são apresentados. O motor aberto é de um antigo *driver* de disco de 5 ¼ de um computador e o outro o motor de uma impressora jato de tinta.

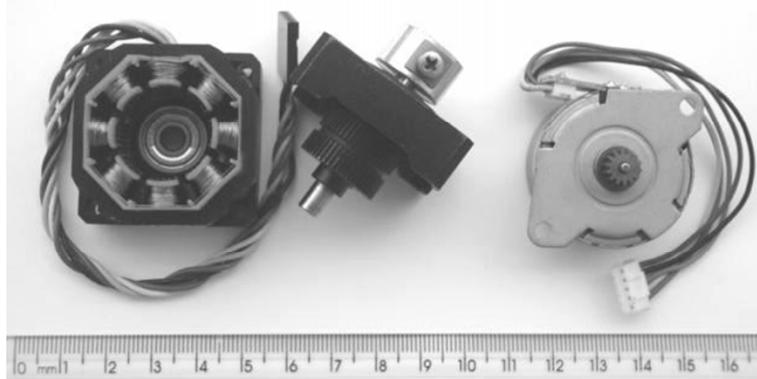


Fig. 9.21 – Motores de passo unipolar.

Na fig. 9.22a, é ilustrada a estrutura de um motor de passo unipolar de 4 fases; na fig. 9.22b, uma forma mais simplificada de representação. O princípio de funcionamento do motor de passo é simples: de acordo com as bobinas energizadas, o rotor irá girar a passos discretos, alinhando seu campo magnético com o campo magnético produzido pelas bobinas.

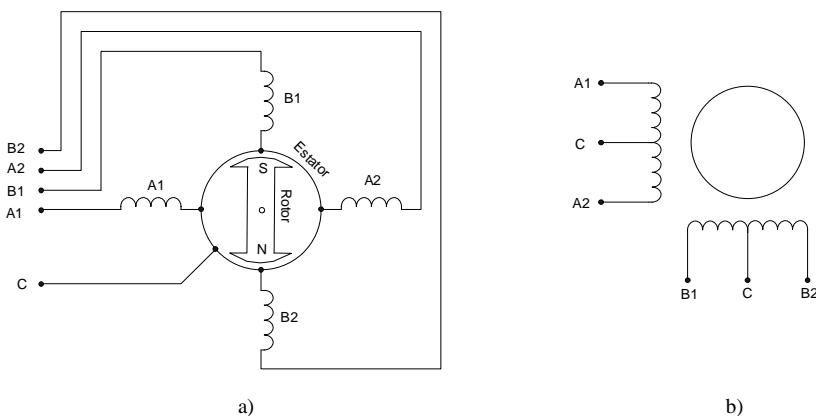


Fig. 9.22 – Controle de um motor de passo unipolar, a) diagrama de um motor de passo de 4 fases, b) diagrama simplificado.

O controle do motor de passo é feito com a sequência correta de energização das bobinas. Podem ser acionadas uma ou mais bobinas ao mesmo tempo. Com o acionamento de duas bobinas simultaneamente, existe a possibilidade de se obter meio passo de giro ou um torque maior. Nas fig. 9.23a e 9.23b, as sequências de energização para um passo por vez são apresentadas, com o emprego de uma e duas bobinas, respectivamente. O acionamento de duas bobinas ao mesmo tempo consome duas vezes mais corrente e, assim, fornece um torque maior. Na fig. 9.24, a sequência para meio passo de rotação é apresentada.

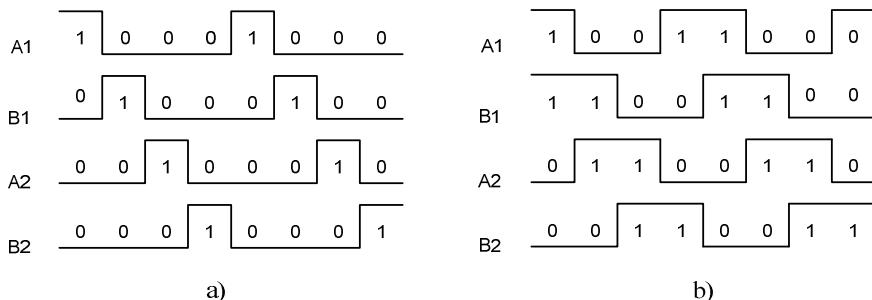


Fig. 9.23 – Sequência de acionamento das bobinas de um motor de passo unipolar para um passo. a) um bobina por vez, b) duas bobinas por vez.

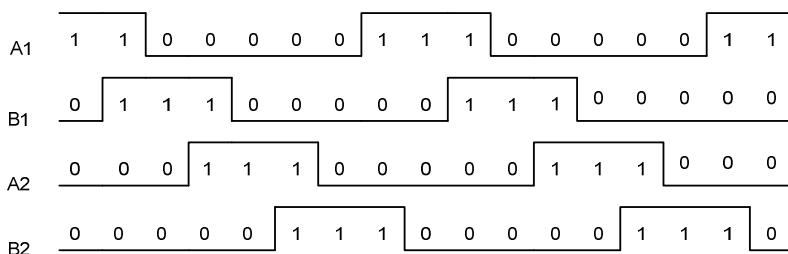


Fig. 9.24 – Sequência de acionamento das bobinas de um motor de passo unipolar para meio passo.

Para o acionamento de um motor de passo, é necessário empregar uma interface (*driver* de corrente) entre o sistema microcontrolado e o

motor para que as correntes exigidas de operação possam ser fornecidas (apêndice D). Geralmente, o ponto comum das bobinas é ligado ao terminal positivo da tensão de alimentação e as bobinas são acionadas quando aterradas.

Outro tipo de motor de passo é o bipolar, que difere do anterior por não conter o ponto comum de ligação entre as bobinas, exigindo, assim, um circuito de controle bem mais complexo.

Exercícios:

9.15 – Elaborar um programa para controlar o motor de passo unipolar da fig. 9.25. Dois botões controlam a direção de rotação e outros dois, a velocidade. Primeiro, deve-se consultar o manual do fabricante do ULN2803 para entender suas características e funcionamento.

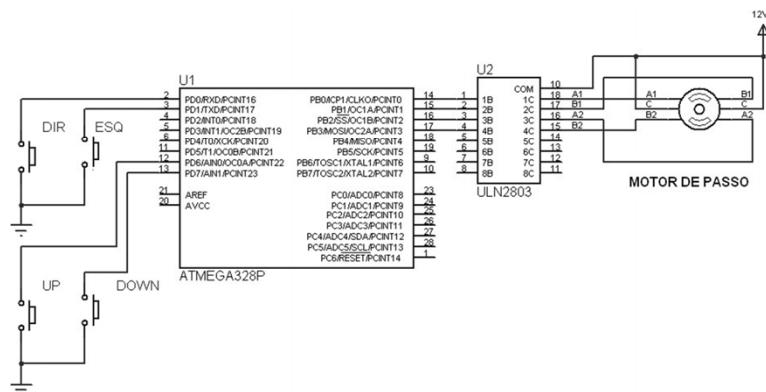


Fig. 9.25 – Controle de um motor de passo unipolar.

9.16 – Como funciona um motor de passos bipolar e como deve ser o seu circuito de controle?

9.8 MÓDULO DE ULTRASSOM - SONAR

Com a crescente popularização da eletrônica, foram disponibilizados módulos sensores prontos para serem utilizados com microcontroladores; um deles, é o de ultrassom. Eles geram um sinal sonoro acima da percepção humana (que é de 20 kHz) e determinam o tempo que esse sinal leva para alcançar e retornar ao encontrar um obstáculo, permitindo dessa forma, determinar a distância do sensor ao obstáculo. Na fig. 9.26, é apresentado um módulo desses, usualmente encontrado no mercado mundial, o HC-SR04. Esse módulo possui toda a eletrônica necessária ao seu funcionamento, bastando ao microcontrolador ordenar a medição da distância e interpretar o sinal de retorno. Possui 4 pinos, dois de alimentação (5 V e GND), um para o acionamento do sensor e outro para a leitura do sinal que indica a distância do objeto. Na fig. 9.27, os sinais de trabalho do módulo são apresentados.

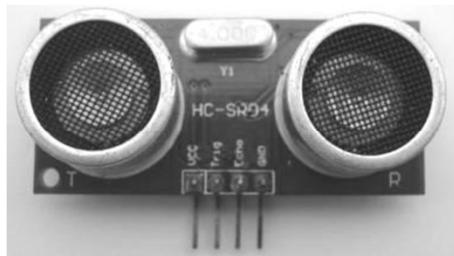


Fig. 9.26 – Módulo de ultrassom HC-SR04.

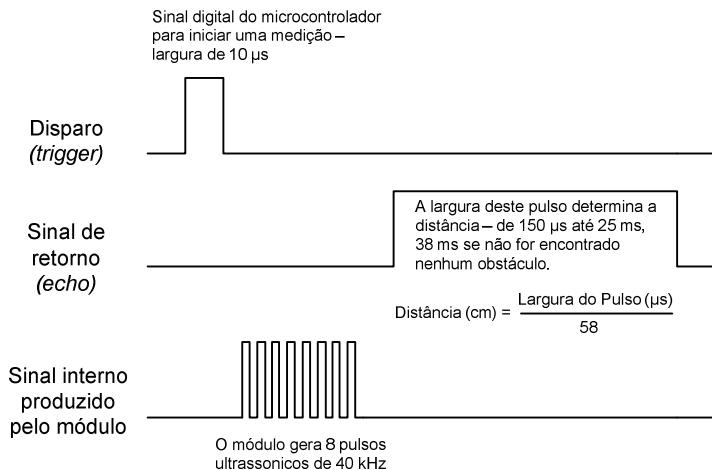


Fig. 9.27 – Sinais de trabalho do módulo HC-SR04.

É necessário enviar um pulso de 10 µs ao módulo para iniciar uma medição. Após isso, o módulo irá gerar 8 pulsos ultrassônicos de 40 kHz para avaliar a distância dele em relação ao objeto, produzindo um sinal de retorno, cuja largura indica a distância do sensor ao objeto. Esse valor estará compreendido entre 150 µs e 25 ms, caso nenhum obstáculo seja encontrado a largura do pulso será de 38 ms. O tempo entre as conversões deve ser de no mínimo 50 ms. O fabricante do módulo especifica que a distância medida deve estar compreendida entre 2 cm e 5 m, que a medição apresenta uma resolução de 0,3 cm e que o ângulo ativo de medição é de 15°.

Dadas as características do sinal retornado pelo módulo de ultrassom, para o ATmega328, o TC1 é o mais adequado para medir o período desse sinal, como exemplificado pelo programa a seguir. Na fig. 9.28 é apresentado o circuito empregado.

Sonar.c

```
//===================================================================== //
//          Programa para teste do módulo Sonar HC-SR04           //
//===================================================================== //
#include "def_principais.h"      //inclusão do arquivo com as principais definições
#include "LCD.h"

#define DISPARO PB1

unsigned int Inicio_Sinal, Distancia;
prog_char mensagem1[] = "Distanc =      cm\0";
prog_char mensagem2[] = "xxx\0";
//-----
ISR(TIMER1_CAPT_vect)          //interrupção por captura do valor do TCNT1
{
    cpl_bit(TCCR1B,ICES1);    //troca a borda de captura do sinal

    if(!tst_bit(TCCR1B,ICES1))//lê o valor de contagem do TC1 na borda de subida do sinal
        Inicio_Sinal = ICR1;//salva a primeira contagem para determinar a largura do pulso
    else                      //lê o valor de contagem do TC1 na borda de descida do sinal
        Distancia = (ICR1 - Inicio_Sinal)/58; /*agora ICR1 tem o valor do TC1 na borda de
                                                 descida do sinal, então calcula a distância */
}
//-----
int main()
{
    unsigned char digitos[tam_vetor];//declaração da variável para armazenagem dos dígitos
    DDRD = 0xFF;
    DDRB = 0b00000010;//somente pino de disparo como saída (PB1), captura no PB0 (ICP1)
    PORTB = 0b11111101;

    TCCR1B = (1<<ICES1)|(1<<CS11); //TC1 com prescaler = 8, captura na borda de subida
    TIMSK1 = 1<<ICIE1;             //habilita a interrupção por captura
    sei();                         //habilita a chave de interrupções globais

    inic_LCD_4bits();
    escreve_LCD_Flash(mensagem1);

    while(1)
    {
        //pulso de disparo
        set_bit(PORTB,DISPARO);
        _delay_us(10);
        clr_bit(PORTB,DISPARO);

        cmd_LCD(0x8A,0);
        if(Distancia<431)//se o pulso for menor que 25 ms mostra o valor da distância
        {
            ident_num(Distancia, digitos);
            cmd_LCD(digitos[2],1);
            cmd_LCD(digitos[1],1);
            cmd_LCD(digitos[0],1);
        }
        else //senão escreve xxx no valor
            escreve_LCD_Flash(mensagem2);

        _delay_ms(50);//mínimo tempo para uma nova medida de distância
    }
}
//=====================================================================
```

Os arquivos de inclusão no início do código foram apresentados no capítulo 5. Entretanto, no arquivo **def_principais.h** foi incluída a biblioteca para tratamento das interrupções (#include <avr/interrupt.h>).

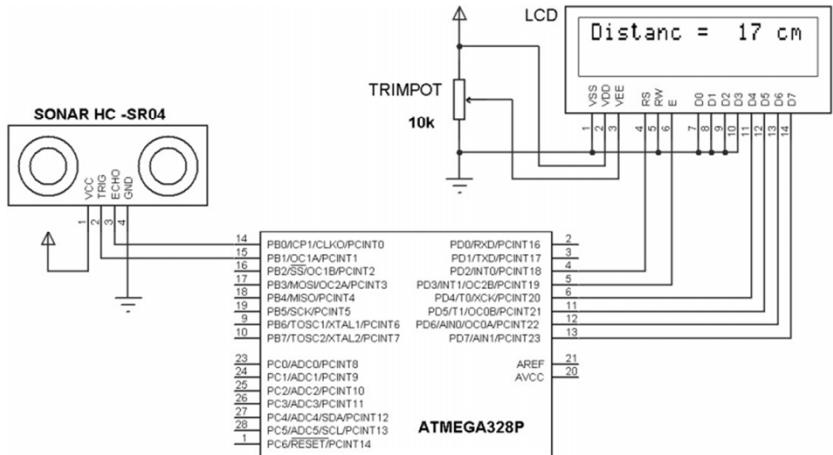


Fig. 9.28 – Circuito para teste do sonar.

Para medir a largura do pulso gerado pelo módulo, utiliza-se a interrupção por captura do TC1. Quando o sinal na entrada do pino ICP1 passa de 0 para 1 (borda de subida), é gerado um pedido de interrupção, onde se armazena o valor de contagem do TC1. Então, altera-se a interrupção por captura para ser gerada na borda de descida do sinal. Quando essa ocorre, o novo valor do TC1 é subtraído do primeiro, que foi coletado na borda de subida. Dessa forma, determina-se quantas contagens foram realizadas no ciclo ativo do sinal e pode-se avaliar a largura do sinal retornado pelo sensor.

Os testes práticos mostraram que o sonar utilizado não é muito preciso, porém apresenta um estimativa razoável da distância.

Exercício:

9.17 – Qual a resolução temporal empregada para o TC1 do programa de trabalho com o módulo sonar HC-SR04? Qual sua relação com a resolução do módulo?

Por que o limite para a apresentação da distância foi de 431 cm?

10. GERANDO MÚSICAS COM O MICROCONTROLADOR

Neste capítulo, uma técnica para a reprodução de sons monofônicos pelo microcontrolador AVR é apresentada. Essa técnica permite a reprodução de várias músicas, com pequenas alterações no programa do microcontrolador, e cujas cifras musicais podem ser encontradas na internet.

Uma aplicação interessante para sistemas microcontrolados é a geração de pequenas músicas (sons monofônicos), reproduzidas em um pequeno alto-falante (*buzzer*), como aqueles feitos para serem soldados diretamente em placas de circuito impresso. Existe dois tipos de alto-falantes que podem ser acionados diretamente por um microcontrolador: os piezelétricos e os magnéticos. Os piezelétricos geram o som quando uma tensão elétrica faz vibrar uma placa cerâmica piezelétrica conectada a uma placa metálica. Os magnéticos são uma versão miniatura dos alto-falantes comuns, possuem um imã e uma bobina; quando a tensão é aplicada a bobina interage com o imã e faz vibrar uma lâmina metálica que produz o som.

Os *buzzers* podem possuir internamente um circuito oscilador (*internal driver*), bastando apenas alimentá-los para a produção do som; esses são conhecidos como *buzzers* indicadores, cuja frequência do som é fixa. Contrariamente, existem os *buzzers* transdutores, que não possuem esse circuito; nesse caso, é necessário gerar uma onda quadrada para fazê-los produzir o som (*external driver*). Esse segundo tipo de *buzzer* deve ser utilizado para criar pequenas músicas com um microcontrolador. Assim, através de um sinal digital com frequência adequada, é possível gerar as notas musicais na sequência dada por uma cifra musical.

Na fig. 10.1, é apresentado um circuito microcontrolado empregando um pequeno *buzzer* transdutor magnético. Poderia ser empregado um alto-

falante comum, mas nesse caso, dependendo da potência e qualidade sonora de saída, seriam necessários circuitos adicionais, como um filtro para o sinal gerado pelo microcontrolador e um amplificador de saída.

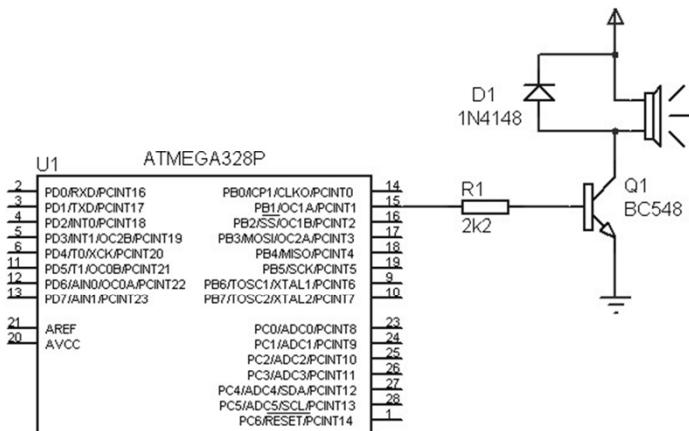


Fig. 10.1 – Circuito microcontrolado para a geração de músicas.

Em resumo, uma música será dada por um conjunto de sinais de frequência e duração variável. Como o sinal é digital, a amplitude do sinal será sempre constante. O desafio para quem não entende muito de música é converter as notas musicais de uma determinada música nos seus valores de frequência e tempo de duração. Esse problema é resolvido usando a codificação empregada pela Nokia nos seus telefones celulares. Essa codificação é utilizada para produzir os conhecidos *ringtones*, cuja codificação é representada por um texto no formato ASCII, conhecida por RTTTL (*Ring Tones Text Transfer Language*). Existem centenas de *ringtones* com esse formato para download gratuito na internet.

Um arquivo RTTTL é um arquivo de texto, sem espaço entre os caracteres, contendo o nome do *ringtone*, uma seção de controle e uma seção contendo uma sequência de comandos separados por vírgulas. Por exemplo, para um arquivo com o nome Simpsons:

```
Simpsons:d=4,o=5,b=160:32p,c.6,e6,f#6,8a6,g.6,e6,c6,8a,8f#,8f#,8f#,2g
```

A codificação funciona da seguinte forma:

<nome> : <seção de controle> : <comando das notas musicais>

Sendo que:

- **<nome>** é uma *string* com comprimento máximo de 10 caracteres.
- **<seção de controle>** possui o **<nome do controle> = <valor>**, cujos controles são dados por **o**, **d**, e **b**. O controle **o** representa a escala *default* das notas musicais (oitava), **d**, a duração *default* de cada nota musical e **b**, as batidas por minuto (dita o ritmo da música). Se não especificados, esses valores são: **d = 4**, **o = 6** e **b = 63**.
- **<comando das notas musicais>** possui a informação da duração da nota musical, qual nota será tocada, duração especial, se houver, e a escala. Assim, a informação de uma nota musical é representada por:

<duração> <nota> <duração especial> <escala>,

- **<duração>** pode ser **1, 2, 4, 8, 16 ou 32**. Seu valor é o divisor de uma nota musical completa. Desta forma:

1/1 = nota completa – semibreve (possui 4 batimentos).

1/2 = meia nota – mínima (possui 2 batimentos).

1/4 = um quarto de nota – semínima (possui 1 batimento).

1/8 = um oitavo de nota – colcheia (possui $\frac{1}{2}$ batimento).

1/16 = um dezesseis avos de nota – semicolcheia (possui $\frac{1}{4}$ de batimento).

1/32 = um trinta e dois avos de nota – fusa (possui $\frac{1}{8}$ de batimento).

- **<nota>** pode ser **p, c, c#, d, d#, e, f, f#, g, g#, a, a#, b, (p** significa pausa e as demais letras são as notas musicais). Uma nota pode ser seguida do caractere de **<duração especial> ‘.’** (ponto), que significa uma nota com duração 50% maior que o normal.
- **<escala>** pode ser **4, 5, 6, 7**, correspondente à escala da oitava empregada: **4 = 440 Hz, 5 = 880 Hz, 6 = 1,76 kHz, 7 = 3,52 kHz** (ver a tab. 10.1). Eventualmente, a escala pode ser seguida pelo caractere de duração especial.

Qualquer controle desconhecido deve ser ignorado na interpretação do arquivo RTTTL.

No ATmega328, as frequências são geradas com o uso de algum TC no modo CTC. Por permitir uma melhor precisão, o TC1 de 16 bits é o mais indicado. Considerando o uso de uma frequência de trabalho de 16 MHz e com o emprego da eq. 9.8, calculam-se os valores de OCR1A do TC1 para gerar as frequências da escala musical. Os resultados, calculados com arredondamento para o número inteiro mais próximo, são apresentados na tab. 10.1. É importante que a frequência de trabalho para o microcontrolador seja exata. Dessa forma, deve ser empregado um sinal de *clock* preciso para a CPU, tal como o obtido com o uso de um cristal externo.

Tab. 10.1 – Valores para o OCR1A (16 MHz) no controle da frequência gerada pelo TC1 empregando o modo CTC (4° - 7° oitava), eq. 9.8 com *prescaler* = 1.

NOTA	4° [Hz] → OCR1A	5° [Hz] → OCR1A	6° [Hz] → OCR1A	7° [Hz] → OCR1A
a (Lá)	440,0 → 18181	880,0 → 9090	1760,0 → 4544	3520,0 → 2272
a# (Lá#)	466,2 → 17159	932,4 → 8579	1864,7 → 4289	3729,4 → 2144
b (Si)	493,9 → 16197	987,8 → 8098	1975,7 → 4048	3951,3 → 2024
c (Dó)	523,3 → 15287	1046,6 → 7643	2093,2 → 3821	4186,5 → 1910
c# (Dó#)	554,4 → 14429	1108,8 → 7214	2217,7 → 3606	4435,5 → 1803
d (Ré)	587,4 → 13618	1174,8 → 6809	2349,7 → 3404	4699,5 → 1701
d# (Ré#)	622,4 → 12852	1244,8 → 6426	2489,5 → 3212	4979,1 → 1606
e (Mi)	659,4 → 12131	1318,8 → 6065	2637,7 → 3032	5275,3 → 1516
f (Fá)	698,7 → 11449	1397,3 → 5724	2794,6 → 2862	5589,2 → 1430
f# (Fá#)	740,2 → 10807	1480,4 → 5403	2960,8 → 2701	5921,8 → 1350
g (Sol)	784,3 → 10199	1568,2 → 5100	3137,1 → 2549	6274,1 → 1274
g# (Sol#)	830,9 → 9627	1661,9 → 4813	3323,7 → 2406	6647,4 → 1202

Um programa para ler um arquivo RTTTL necessita analisar os caracteres do arquivo em ordem sequencial e ajustar as frequências geradas, bem como o tempo de duração das notas, de acordo com as informações lidas.

Para simplificar a troca de oitavas e economizar memória, a primeira oitava (4°) é a inicial; caso se deseje trocá-la, basta dividir os valores de OCR1A por um múltiplo de 2.

Para determinar o tempo que cada nota musical ficará ativa, toma-se a menor nota para determinar a base de tempo ($1/32 = \text{fusa}$). Assim, com o número de batimentos da música por minuto, calcula-se a fração de tempo que formará todas as notas musicais.

O número de batimentos por segundo de uma música é o número de batimentos por minuto dividido por 60:

$$bps = \frac{bpm}{60} \quad (10.1)$$

Sabendo-se que uma nota completa (semibreve) possui 4 batimentos, o número de notas completas por segundo é dado por:

$$\text{Semibreves} = \frac{bps}{4} = \frac{bpm}{240} \quad (10.2)$$

Então, o tempo de uma única semibreve é dado por:

$$\Delta t_{\text{semibreve}} = \frac{240}{bpm} \quad (10.3)$$

Assim, a fração $1/32$ de uma semibreve será dada por:

$$\Delta t_{\text{fusa}} = \frac{240}{bpm \times 32} = \frac{7,5}{bpm} \quad (10.4)$$

Com o tempo de uma nota fusa, computa-se o tempo das demais notas musicais.

Na fig. 10.2, apresenta-se o fluxograma de um programa para a leitura de um arquivo RTTTL. Na sequência, é apresentado o programa para o ATmega328. O tempo que o programa gasta (leitura da nota) até chegar no tempo em que a nota musical ficará ativa é desprezável. Para substituir a

música, basta copiar e colar o texto do arquivo RTTTL na variável MUSICA e corrigir os dados de controle nas variáveis declaradas.

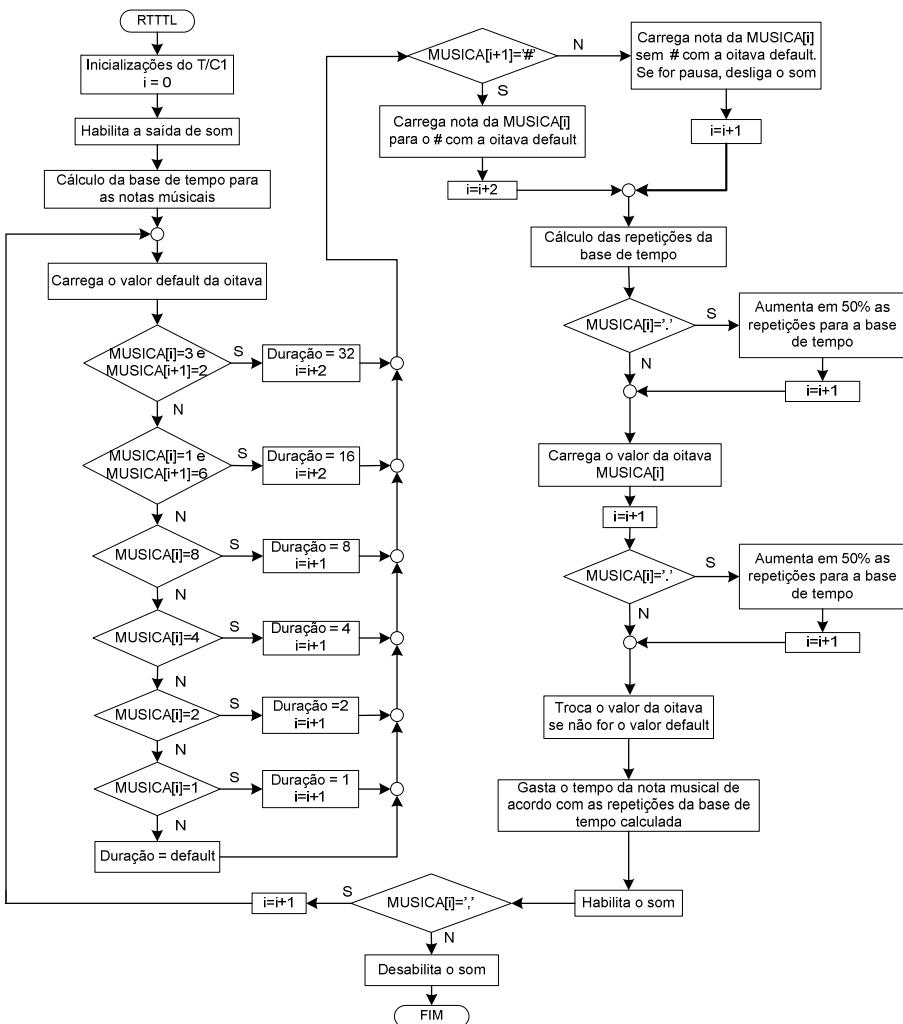


Fig. 10.2 – Fluxograma de um programa para a leitura de um arquivo RTTTL.

Musica.c

```
//===================================================================== //
//          MUSICAS COM O ATMEGA                                //
//          Leitura de arquivos RTTTL - Ringtones                //
//===================================================================== //
#define F_CPU 16000000UL           //frequência de operação de 16MHz
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <util/delay.h>

//Definições de macros
#define set_bit(Y,bit_X) (Y|=(1<<bit_X))    //ativa bit
#define clr_bit(Y,bit_X) (Y&=~(1<<bit_X))   //limpa bit
#define som      PB1 //pino OC1A para saída do sinal

#define d_inic     4 //valor inicial de duração da nota musical
#define o_inic     5 //valor inicial da oitava
#define b        100 /*o nr de batidas indica a velocidade da musica (alterar
                     para mudar a velocidade), maior = mais rápido*/
#define t_min  (7500/b)*10//tempo mínimo para formar o tempo base das notas musicais (1/32)
/*Max. de 255 caract. para MUSICA, caso contrário o índice de varredura deve ser
alterado para unsigned int (não usar espaço entre os caracteres)*/

//Marcha Imperial - Star Wars
const char MUSICA[] PROGMEM
= {"e,e,e,8c,16p,16g,e,8c,16p,16g,e,p,b,b,b,8c6,16p,16g,d#,8c,16p,16g,e,8p"};
//------------------------------------------------------------------------------

int main()
{
    unsigned int k;
    unsigned char d, o, j, n, i=0;

    DDRB |= (1<<som);           //habilita a saída de som

    TCCR1A = 1<<COM1A0;//TC1 modo CTC comparação com OCR1A, prescaler=1
    TCCR1B = (1<<WGM12)|(1<<CS10);
    //-----//
    //LEITURA E EXECUÇÃO DO ARQUIVO RTTTL
    //-----//
    do
    {
        o = o_inic;           //carrega o valor default para a oitava

        if((pgm_read_byte(&MUSICA[i])=='3')&&(pgm_read_byte(&MUSICA[i+1])=='2')){d=32; i+=2;}
        else if((pgm_read_byte(&MUSICA[i])=='1')&&(pgm_read_byte(&MUSICA[i+1])=='6')){d=16;
                                         i+=2;}

        else if(pgm_read_byte(&MUSICA[i])=='8') {d=8; i++;}
        else if(pgm_read_byte(&MUSICA[i])=='4') {d=4; i++;}
        else if(pgm_read_byte(&MUSICA[i])=='2') {d=2; i++;}
        else if(pgm_read_byte(&MUSICA[i])=='1') {d=1; i++;}
        else d=d_inic; //carrega o valor default para a duração

        if(pgm_read_byte(&MUSICA[i+1])=='#')
        {
            switch(pgm_read_byte(&MUSICA[i]))//carrega a oitava # default (4a)
            {
                case 'a': OCR1A = 17159; break;//A# - Lá#
                case 'c': OCR1A = 14429; break;//C# - Dó#
                case 'd': OCR1A = 12852; break;//D# - Ré#
                case 'f': OCR1A = 10807; break;//F# - Fá#
                case 'g': OCR1A = 9627; break;//G# - Sól#
            }
            i+=2;
        }
    }
}
```

```

else
{
    switch(pgm_read_byte(&MUSICA[i]))//carrega a oitava default (4a)
    {
        case 'a': OCR1A = 18181; break;//A - Lá
        case 'b': OCR1A = 16197; break;//B - Si
        case 'c': OCR1A = 15287; break;//C - Dó
        case 'd': OCR1A = 13618; break;//D - Ré
        case 'e': OCR1A = 12131; break;//E - Mi
        case 'f': OCR1A = 11449; break;//F - Fá
        case 'g': OCR1A = 10199; break;//G - Sol
        case 'p': clr_bit(TCCR1A,COM1A0); break;//p = pausa
    }
    i++;
}
n = 32/d;           //tempo de duração de cada nota musical
if(pgm_read_byte(&MUSICA[i])=='.'){n=n+(n/2); i++;}//duração 50% >
if(pgm_read_byte(&MUSICA[i])=='4') { o=4; i++; }
else if(pgm_read_byte(&MUSICA[i])=='5'){ o=5; i++; }
else if(pgm_read_byte(&MUSICA[i])=='6'){ o=6; i++; }
else if(pgm_read_byte(&MUSICA[i])=='7'){ o=7; i++; }
if(pgm_read_byte(&MUSICA[i])=='.'){n=n+(n/2); i++;}// duração 50% >
switch(o)//troca a oitava se não for a default (o = 4)
{
    case 5: OCR1A = OCR1A>>1; break; //divide por 2
    case 6: OCR1A = OCR1A>>2; break; //divide por 4
    case 7: OCR1A = OCR1A>>4; break; //divide por 8
}
//-----
for(j=0;j<n;j++)//nr de repetições para a nota 1/32
{
    for(k=t_min;k!=0;k--) _delay_us(100);
}
//-----
set_bit(TCCR1A,COM1A0); //habilita o som
}while(pgm_read_byte(&MUSICA[i++])==',');//leitura até o final da música
TCCR1A=0;           //desabilita o som e o TC1
TCCR1B=0;
//-----
while(1{})      //laço infinito
}
//=====================================================================

```

Exercícios:

10.1 – Utilizando o modo de geração de frequência do ATmega328, faça um programa para gerar as notas musicais básicas de um teclado com 1 oitava (pode ser a 4^a). Empregue um circuito adequado, incluindo as teclas. Na fig. 10.3, é ilustrada a organização das notas musicais no teclado (para aumentar o teclado basta colocar outra oitava em série com a primeira). Obs.: só pode ser tocada uma nota por vez.

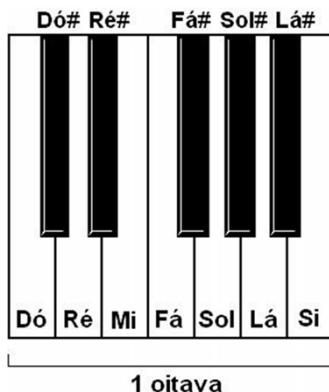


Fig. 10.3 – Organização de uma oitava em um teclado musical.

10.2 – Com base no programa apresentado neste capítulo, faça um programa para aumentar e diminuir a velocidade de uma pequena música (batida) no ATmega328, toda vez que um botão for pressionado. Gere uma função leitora de arquivos RTTTL e utilize uma opção para a seleção de algumas músicas. Pesquise por músicas no formato RTTTL na internet.

10.3 – Otimize o programa de leitura de um arquivo RTTTL para gerar a base de tempo para as notas musicais utilizando o TC2.

11. TÉCNICAS DE MULTIPLEXAÇÃO

Neste capítulo, algumas técnicas importantes de multiplexação são apresentadas. Elas são utilizadas para diminuir o número de componentes externos ao microcontrolador e/ou número de I/Os necessários. Trata-se de técnicas de multiplexação de sinais para emprego em *displays*, acionamento de conjuntos de LEDs (matriz) e outros dispositivos eletrônicos.

Quando são necessários vários pinos de I/O para o acionamento de um determinado circuito e o microcontrolador não os dispõem, é fundamental o emprego da multiplexação: técnica para transitar com vários dados em uma mesma via ou barramento. A multiplexação também é empregada para diminuir o número de vias e pode diminuir a complexidade física das placas de circuito impresso.

A ideia da multiplexação é dividir as atividades no tempo, empregando o mesmo meio físico para isso. A desoneração do hardware é substituída por um aumento na complexidade do software de controle e no tempo de execução das tarefas. Entretanto, devido à considerável velocidade de processamento dos sistemas envolvidos, geralmente isto não é um problema. As melhores técnicas de multiplexação empregam o menor número possível de componentes externos para cumprir as funções que devem ser desempenhadas pelo hardware.

11.1 EXPANSÃO DE I/O MAPEADA EM MEMÓRIA

Uma das principais técnicas de multiplexação empregada em sistemas microprocessados é ilustrada na fig. 11.1. Existem dois barramentos: um de dados e um de endereços. Cada dispositivo ligado aos barramentos responde a um ou mais endereços exclusivos e comunica-se com o microprocessador através do barramento de dados. Essa técnica é denominada expansão de I/O mapeada em memória.

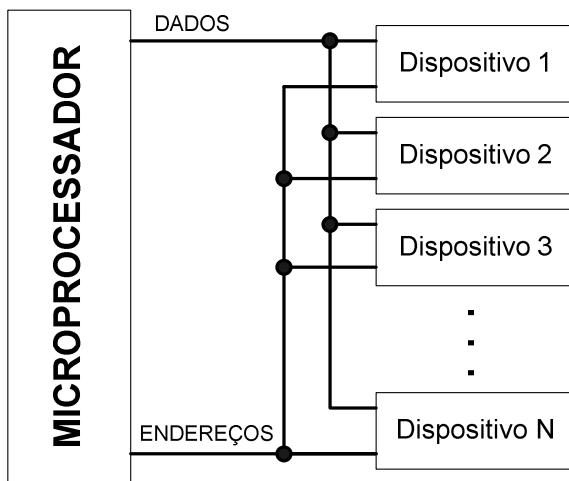


Fig. 11.1 – Controle de vários dispositivos em um sistema microprocessado.

O número de dispositivos diferentes que podem ser ligados ao barramento de dados depende do número de bits do barramento de endereços e é expresso por uma potência de 2. Por exemplo, se o barramento de endereços for de 6 bits, podem ser endereçados 64 dispositivos diferentes (2^6).

Se houver necessidade de escrita e leitura no dispositivo, pode-se empregar o bit menos significativo do seu endereço para essa informação. Supondo que um determinado dispositivo responda ao endereço 0b10101x,

quando o bit menos significativo desse endereço for 0, a operação pode ser de escrita no dispositivo e quando for 1, o dispositivo pode escrever no barramento de dados. Nesse caso, com 6 bits, poderiam ser endereçados 32 dispositivos diferentes (2^{6-1}).

No sistema da fig. 11.1, cada dispositivo é responsável pela decodificação do seu endereço e não deve responder a nenhum endereço diferente do seu. Assim, quando não estiver sendo acessado, o barramento de dados do dispositivo deve se portar como uma alta impedância, caso contrário, haverá colisão de dados (curtos-circuitos).

Na fig. 11.2, é apresentado um exemplo de multiplexação em que 32 entradas e 32 saídas são controladas com 12 pinos do ATmega328. O CI 74238 decodifica o endereço de entrada (pinos A, B e C), habilitando um único pino de saída (Y0-Y7); com 3 bits de entrada, tem-se, então, 8 saídas independentes (2^3). Nessas, os quatro bits menos significativos habilitam 4 *latches*, os CIs 74373 (saídas). Cada *latch* habilitada colocará os dados do barramento de dados na saída e os manterá lá até a ocorrência de um novo sinal de habilitação. Na parte mais significativa dos bits de saída do CI 74238, encontram-se 4 *buffers tri-state* (CIs 74244). Esses CIs quando habilitados, transferem os dados de suas entradas para o barramento de dados, e quando desabilitados, mantém suas saídas em alta impedância. O microcontrolador escreve o endereço do CI que deseja acessar no barramento de endereços, e faz sua escrita ou leitura. O pino de habilitação (E1) do 74238 foi empregado para desabilitar todos os CIs quando desejado.

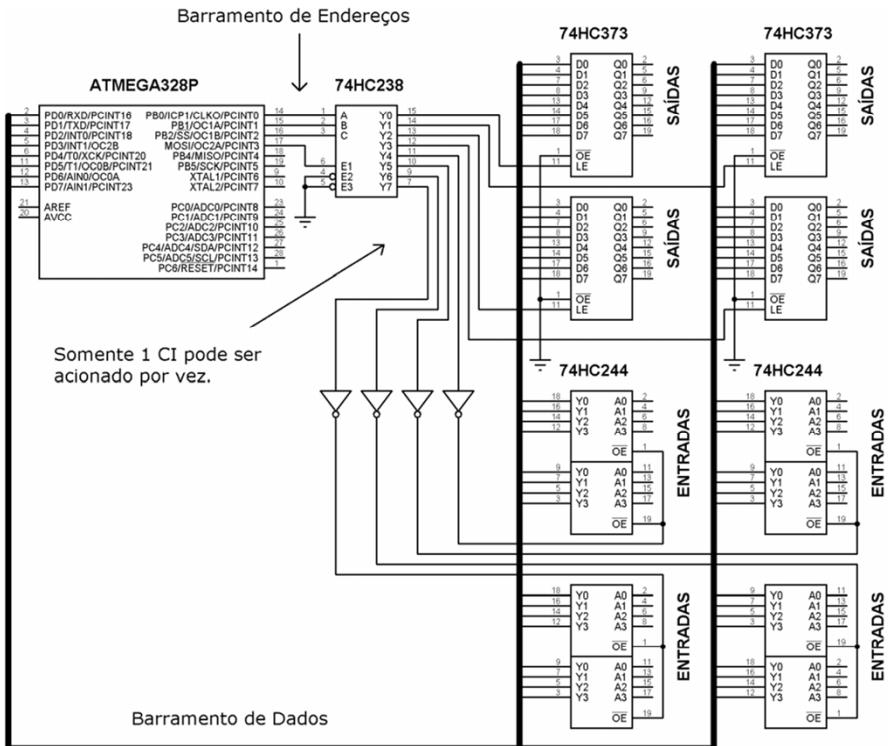


Fig. 11.2 – Exemplo de um sistema multiplexado com 32 saídas e 32 entradas, empregando 12 vias de controle.

Com o circuito da fig. 11.2, houve uma expansão considerável de entradas e saídas do microcontrolador. Com um barramento de dados de 8 bits e um de endereços de 3 bits conseguiu-se, com componentes externos, uma expansão para 32 saídas e 32 entradas.

Outra possibilidade para utilizar menos pinos de I/O do microcontrolador no endereçamento de dispositivos externos é o uso de um contador de década, que ativa uma de suas saídas a cada pulso de *clock* na sua entrada. Na fig. 11.3, o circuito que ilustra esse conceito é apresentado. O contador 4017 poderia substituir o 74238 da fig. 11.2 realizando uma contagem contínua de 0 até 7. Agora, poderiam ser acessados os mesmos I/Os que no circuito da fig. 11.2, com uma

diminuição de 2 pinos do barramento de dados. Entretanto, dessa forma, não é mais possível acessar qualquer um dos dispositivos do barramento aleatoriamente (com um endereço), é necessário realizar o acesso sequencial deles através dos pulsos de *clock* enviados ao 4017.

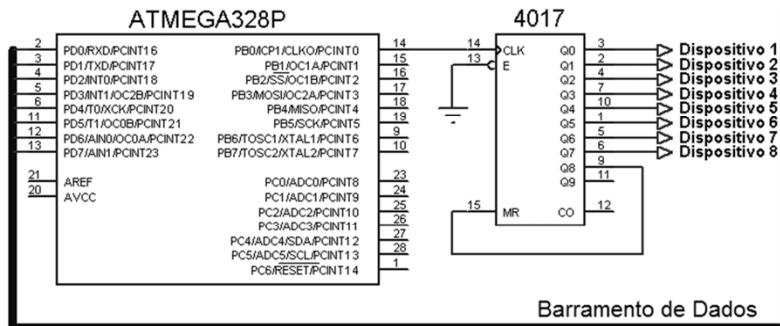


Fig. 11.3 – Sistema multiplexado com 32 saídas e 32 entradas, empregando 9 vias de controle.

11.2 CONVERSÃO SERIAL-PARALELO

Outra técnica para a expansão de I/Os é a conversão de dados seriais para paralelos, servindo para aumentar o número de saídas de um sistema. Muitos projetos necessitam dessa forma de expansão de I/O para cumprirem eficientemente suas funções com um mínimo de componentes externos. O custo no uso dessa técnica é um aumento da complexidade do software de controle e uma diminuição da velocidade de processamento devido aos componentes envolvidos.

Circuitos integrados interessantes para a conversão serial-paralelo são o 74595 e o 4094 (*shift registers*). Eles possuem saídas com alta impedância e *latches* internos para que os dados fiquem armazenados nas saídas, permitindo a ligação dos CIs em série (cascateamento). Na fig. 11.4, são apresentados três 4094 ligados em cascata, nos quais a saída de dados de um é ligada na entrada de dados do outro e, assim, sucessivamente. O microcontrolador colocará 24 bits de dados seriais no pino PB0, seguidos

por 24 pulsos de *clock* no pino PB1. A cada pulso de *clock*, o bit de entrada do primeiro 4094 empurra o bit de entrada anterior na fila, preenchendo os registradores internos dos CIs em cascata. Funciona semelhante a uma fila: a entrada de um bit desloca o próximo e, assim, sucessivamente até o preenchimento de todas as vagas. Depois que os 4094 foram preenchidos, é gerado o sinal de *strobe* (pulso) no pino PB2, o que resulta na transmissão dos dados internos dos *latches* para os pinos de saída, os quais ficam presos lá até um novo sinal de *strobe*. Em resumo, preenchem-se todos os *shift registers* e depois se habilita a saída dos seus dados. Estes ficarão inalterados nas saídas até um novo sinal de *strobe* e nesse intervalo, os *shift registers* podem receber novas informações.

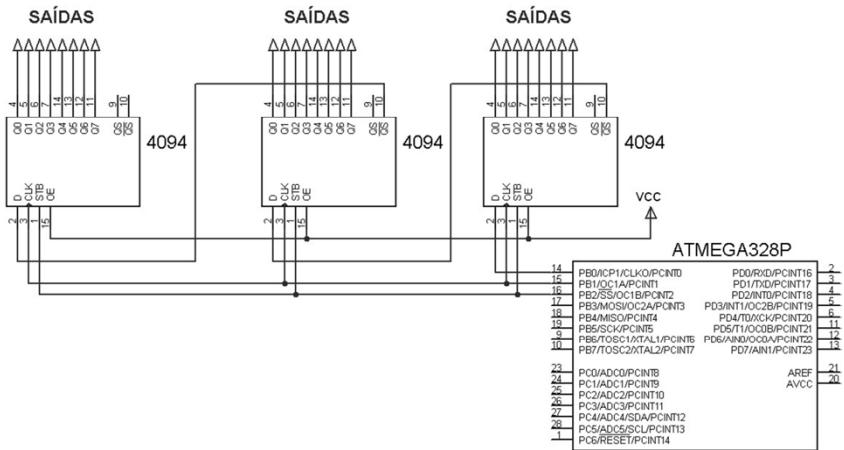


Fig. 11.4 – Emprego do CI 4094 para a conversão serial-paralelo na expansão do número de pinos de saídas de um sistema.

O número de pinos de saída de uma expansão serial-paralelo depende exclusivamente do número de CIs ligados em cascata, visto que o número de pinos de controle será sempre o mesmo.

Na sequência, é apresentado um programa com um sub-rotina exemplificando o envio de 3 bytes para os 4094 de acordo com o circuito da fig. 11.4. O sinal de *strobe* é gerado pelo software de controle para

transferir a informação para as saídas dos 4094. No caso do programa, a sub-rotina de envio é utilizada 3 vezes (envio de 3 bytes) antes da geração do pulso de *strobe*.

Serial_paralelo.c

```
//=====================================================================
// Envio de 3 bytes para o 4094
//=====================================================================
#include "def_principais.h"

#define D      PB0          //pino de dados para o 4094
#define CLK    PB1          //pino clock para o 4094
#define STB    PB2          //pino de strobe para o 4094

#define pulso_CLK() set_bit(PORTB,CLK); _delay_us(10); clr_bit(PORTB,CLK)
#define pulso_STB() set_bit(PORTB,STB); _delay_us(10); clr_bit(PORTB,STB)
//------------------------------------------------------------------------------

// Sub-rotina que envia 1 byte para o 4094 - serial/paralelo
//------------------------------------------------------------------------------

void serial_paral(unsigned char c)
{
    unsigned char i=8;           //envia primeiro o MSB

    do
    { i--;
        if(tst_bit(c,i))       //se o bit for 1, ativa o pino de DADOS
            set_bit(PORTB,D);
        else                   //se não, o zera
            clr_bit(PORTB,D);

        pulso_CLK();

    } while (i!=0);
}

int main(void)
{
    unsigned char j;
    unsigned char Dados[3]= {0x58, 0xF1, 0xAA};

    DDRB = 0b00000111; //pinos PB0:2 como saídas
    PORTB = 0b11111000; //zera saídas

    for(j=0; j<3;j++)
        serial_paral(Dados[j]); //envia os 3 dados para os 4094 (primeiro o 0x58)

    pulso_STB(); /*depois de enviar os 3 dados dá o pulso de Strobe, neste instante os
                   dados passam para as saída*/
    while(1)
    {}           //laço infinito
}
//=====================================================================
```

O arquivo **def_principais.h** foi visto nos capítulos anteriores.

11.3 CONVERSÃO PARALELO-SERIAL

Dual ao sistema serial-paralelo, a conversão de dados paralelo para serial serve para aumentar o número de entradas de um determinado sistema. Na fig. 11.5, é apresentado um circuito exemplo empregando três CIs 74165 conectados em cascata. Quando o sinal de carga (*load*) é aplicado, as informações constantes na entrada dos CIs são transferidas para os seus registradores internos. Após, à medida que os pulsos de *clock* são gerados, os dados são transferidos serialmente para a saída de dados do sistema. Assim, com 24 pulsos de *clock* consegue-se ler os três 74165 da fig. 11.5. O número de entradas depende exclusivamente do número de conversores paralelo-serial conectados em cascata; para o controle são empregados sempre três pinos.

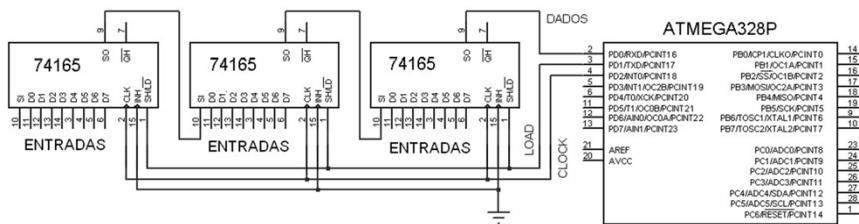


Fig. 11.5 – Emprego do CI 74165 para a conversão paralelo-serial na expansão de entradas de um sistema.

11.4 MULTIPLEXAÇÃO DE DISPLAYS DE 7 SEGMENTOS

Nossos olhos são incapazes de perceber a alteração de um sinal luminoso com frequência acima de 24 Hz, essa característica humana é chamada persistência da visão. Isso significa que se um LED piscar 24 vezes em um segundo, ele aparecerá aos olhos humanos como ligado constantemente. Todavia, um sinal luminoso com frequência de 24 Hz ainda produz cintilação na sua percepção. Desta forma, para uma visualização agradável, emprega-se um sinal de pelo menos 48 Hz.

Sem utilizar a persistência da visão, para ligar diretamente 4 *displays* de 7 segmentos, seriam necessários $8 \times 4 = 32$ pinos do microcontrolador (considerando o uso do ponto do *display*). O problema, além do grande número de pinos, seria a corrente máxima que poderia chegar a 320 mA, no caso de se usar 10 mA por segmento do *display*. Para resolver esse problema, emprega-se a multiplexação temporal baseada na persistência da visão. Na fig. 11.6, é ilustrado um circuito para executar essa função, *displays* anodo comum ligados através de transistores PNP são empregados, ativos quando a base deles é colocada em 0 V (nível lógico 0). Os LEDs são ligados com o mesmo nível de tensão e o circuito de controle drena a corrente deles. O circuito será similar para *displays* catodo comum, nesse caso deve-se empregar transistores NPN, os níveis de ativação serão de 5V (ou nível lógico 1) e o sistema de controle fornecerá corrente aos LEDs dos *displays*.

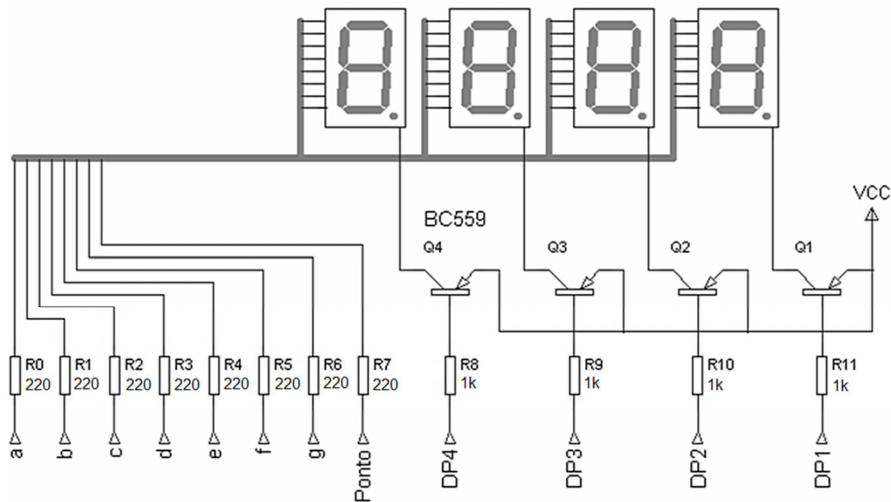


Fig. 11.6 - Acionamento de 4 *displays* de 7 segmentos³⁰.

³⁰ Se o valor de VCC for maior que a tensão de desligamento (nível lógico) na base dos transistores PNP, eles poderão ficar diretamente polarizados e conduzir indevidamente. Transistores PNP funcionam adequadamente quando a tensão de alimentação dos *displays* (VCC) for a mesma do circuito digital de controle, caso contrário, transistores NPN devem ser utilizados.

Num sistema microcontrolado, a multiplexação temporal para *displays* é feita através da chamada varredura. Para os *displays* da fig. 11.6, isso é realizado da seguinte maneira:

1. No início, os *displays* estão apagados e os transistores desabilitados.
2. O dado referente ao primeiro *display* é colocado no barramento.
3. O *display* referente ao dado é ligado, o transistor adequado é acionado.
4. Espera-se o tempo necessário para o acionamento do próximo *display*.
5. Apaga-se o *display* ligado.
6. Coloca-se o dado referente ao próximo *display* da sequência.
7. O *display* é ligado, o transistor adequado é acionado.
8. O processo se repete de forma contínua para cada *display*.

Para que a persistência da visão funcione, cada *display* deve ser ligado com uma frequência mínima de 48 Hz, é usual empregar-se 50 Hz ou mais. Desta forma, considerando-se 4 *displays*, a frequência de varredura do sistema deve ser de 200 Hz (4×50 Hz). Isso significa que cada *display* ficará ligado durante um ciclo da varredura e 3 desligados, resultando em 5 ms de acionamento para cada um ($1/200$). Obviamente, o brilho conseguido com um sistema multiplexado é menor do que em um sistema sem multiplexação. Assim, para se ter uma boa luminosidade é importante acionar os LEDs com a maior corrente possível.

Na programação de um microcontrolador, a forma mais elegante de apresentar a mensagem num conjunto de *displays* é fazer a varredura dentro da rotina de interrupção de algum contador, liberando o programa principal para outras atividades e simplificando o código. Na fig. 11.7, é apresentado o fluxograma de um programa de controle para 4 *displays* empregando a interrupção de um contador. Na sequência, um código

exemplo³¹ desenvolvido para o Arduino, conforme circuito da fig. 11.8 e que apresenta o número 3210 (fig. 11.9).

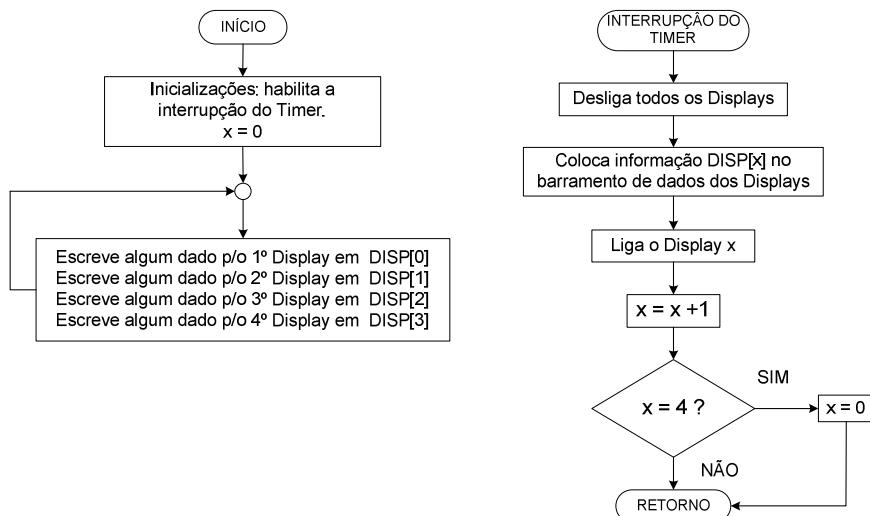


Fig. 11.7 - Fluxograma para o acionamento de 4 *displays* de 7 segmentos.

Varredura_display_7seg.c

```

//=====
//      VARREDURA DE DISPLAYS DE 7 SEGMENTOS
//=====
#define F_CPU 16000000UL

#include <avr/io.h>
#include <avr/interrupt.h>
#define clr_bit(Y,bit_x) (Y&=~(1<<bit_x))

unsigned char DISP[4]; //valores para os displays
//-----
//INTERRUPCAO - VARREDURA DOS DISPLAYS DE 7 SEGMENTOS
//-----
ISR(TIMER0_OVF_vect)
{
    static unsigned char x;

    PORTB |= 0x0F;//apaga todos os displays (o controle dos displays está nos pinos (PB0:PB3)
    PORTD = DISP[x]; //coloca a informação do display no porta correspondente
    clr_bit(PORTB,x); //habilita o display correspondente (PB0:PB3)
    x++;

    if(x==4) x = 0; //após 4 rotações inicializa para o primeiro display
}
//-----

```

³¹ Na seção 5.4, foram apresentados o *display* de 7 segmentos e seus códigos de decodificação (tab. 5.2).

```

int main()
{
    DDRD = 0xFF;           //dados dos displays
    DDRB = 0x0F;           //controle dos displays
    PORTB = 0xFF;          //apaga displays e liga pull-ups
    UCSR0B = 0x00;          //para usar os pinos do PORTD no Arduino
    //TC0 gerando interrupção
    TIMSK0 = 1<<TOIE0;    //habilita a interrupção por estouro do timer 0
    TCCR0B = 1<<CS02;      //CLK/256 prescaler (CLK=16MHz), estouro a cada 4ms
    sei();                 //habilita a interrupção global

    while(1) //qualquer escrita em DISP[x] é automaticamente apresentada nos displays
    {
        DISP[0]= 0xC0; //valor para o número zero
        DISP[1]= 0xF9; //valor para o número um
        DISP[2]= 0xA4; //valor para o número dois
        DISP[3]= 0xB0; //valor para o número três
    }
}
//=====

```

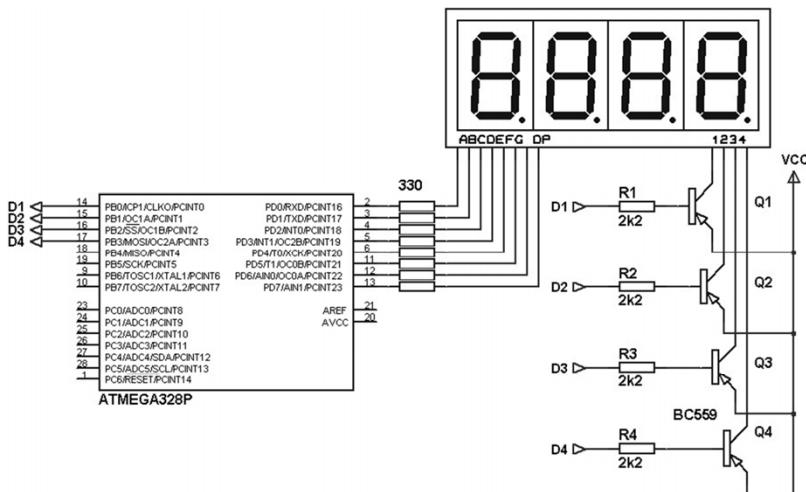


Fig. 11.8 Circuito para o acionamento de 4 *displays* de 7 segmentos anodo comum.



Fig. 11.9- Resultado da escrita multiplexada em 4 *displays* de 7 segmentos.

Exercícios:

- 11.1** – Baseado no exercício 5.12, fig. 5.11a, ligue sequencialmente os 8 LEDs. Comece com uma frequência visível e aumente progressivamente até que os LEDs pareçam estar todos ligados. Qual o tempo que cada LED ficou ligado para a persistência da visão?
- 11.2** – Empregando dois *displays* de 7 segmentos e um botão, desenvolva um programa para o sorteio aleatório e com mesma probabilidade de ocorrência dos números de 1 até 60 (Mega Sena). O número sorteado não deve voltar ao sorteio.
- 11.3** – Elaborar um programa para que o hardware da fig. 11.10 funcione como relógio 24 h. A entrada de sinal para contagem dos segundos é de 60 Hz. O ajuste do horário deve ser feito nos botões específicos.

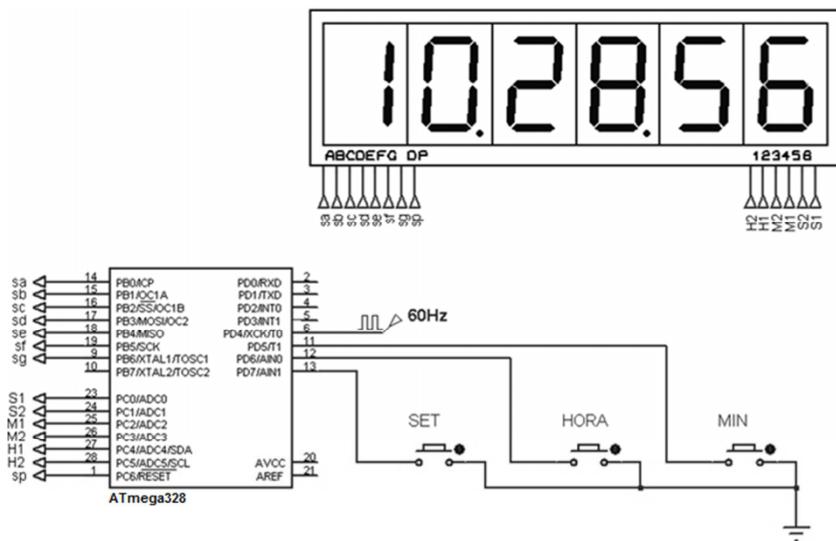


Fig. 11.10 – Relógio com contagem baseado na frequência da rede elétrica (circuito simplificado).

11.5 UTILIZANDO MAIS DE UM LED POR PINO

Como é possível três estados no pino do microcontrolador (0, 1 e alta impedância - entrada sem *pull-up*), um arranjo adequado de LEDs pode resultar em um pino acionando mais de um LED. Na fig. 11.11, é apresentado o circuito para controlar 2 LEDs utilizando apenas um pino de I/O do microcontrolador. Nesse caso, para apagar os LEDs basta colocar o pino em alta impedância; para ligar os dois LEDs em conjunto é necessário empregar a multiplexação temporal, ligando rapidamente cada um deles. Como existe um divisor de tensão entre os resistores R1 e R2, LEDs com uma tensão de trabalho superior à tensão sobre R1 ou R2 não funcionarão adequadamente, pois a tensão fornecida não será suficiente para ligá-los.

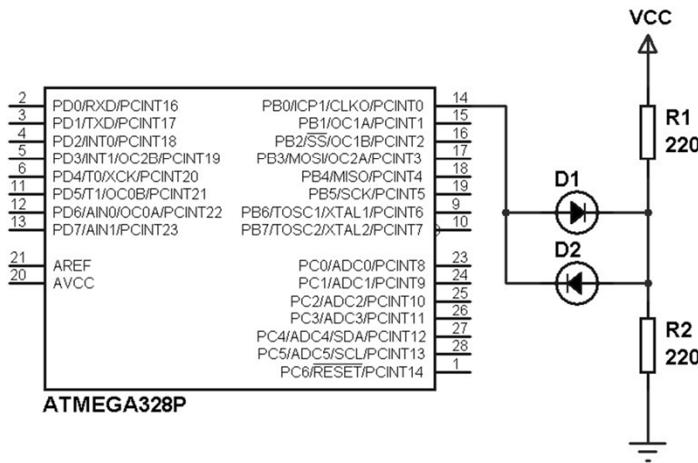


Fig. 11.11 – Empregando 1 pino do microcontrolador para controlar 2 LEDs.

Para ligar 6 LEDs com 3 pinos, pode ser empregado o circuito da fig. 11.12. Na tab. 11.1, é apresentada a configuração que deve ser feita nos pinos do circuito e sua correspondência no controle dos LEDs.

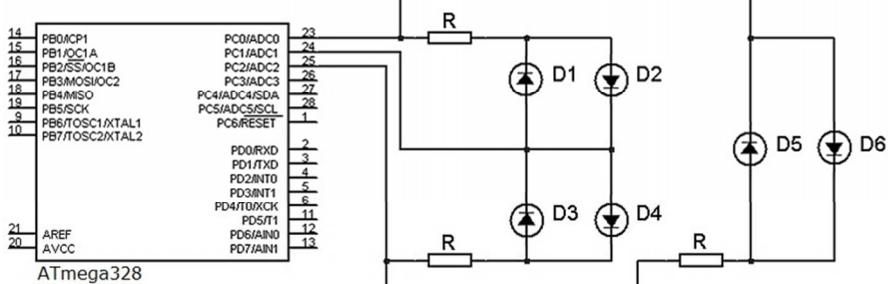


Fig. 11.12 – Empregando 3 pinos do microcontrolador para controlar 6 LEDs.

Tab. 11.1 – Configuração dos pinos PC0..2 da fig. 11.10 e sua correspondência no controle de 6 LEDs (1 indica LED aceso).

Pinos			LEDs					
PC0	PC1	PC2	1	2	3	4	5	6
0	0	0	0	0	0	0	0	0
0	1		1	0	0	0	0	0
1	0		0	1	0	0	0	0
Hi-Z	0	1	0	0	1	0	0	0
Hi-Z	1	0	1	0	0	1	0	0
0	Hi-Z	1	0	0	0	0	1	0
1	Hi-Z	0	0	0	0	0	0	1
0	0	1	0	1	0	0	0	0
0	1	0	1	0	0	1	0	0
0	1	1	1	0	0	0	1	0
1	0	0	0	1	0	1	0	0
1	0	1	0	1	0	0	0	1
1	1	0	0	1	0	0	1	0
1	1	1	0	0	0	0	0	0

Se houver a necessidade do emprego de alguma configuração de LEDs que não conste na tab. 11.1, é necessário empregar a persistência da visão para acionar sequencialmente os LEDs desejados.

O número de LEDs controlados dependerá do número de pinos utilizados e pode ser aumentado usando a mesma lógica acima. O número

de LEDs que podem ser controlados, de acordo com o número de pinos, é dado por:

$$Nr_{LEDs} = Nr_{pinos} \times (Nr_{pinos} - 1) \quad (11.1)$$

Exercício:

11.4 – Baseado no exercício 5.14, crie um dado eletrônico empregando somente 3 pinos para o controle dos LEDs.

11.6 MATRIZ DE LEDs

Um sistema muito comum para a divulgação visual de mensagens publicitárias e informativas é a matriz de LEDs (ver a fig.11.13), onde um conjunto organizado de LEDs, formando um painel de pixels (1 LED = 1 pixel), é comandado por um sistema microcontrolado. Como a quantidade de informação para formar uma imagem é grande, existe a necessidade da multiplexação dos dados e o emprego de algum sistema de varredura (o sistema baseia-se na persistência da visão).

Os LEDs em uma matriz são organizados em linhas e colunas³². Assim, na fig. 11.13, por exemplo, todos os anodos estão conectados à linha que alimenta todo o conjunto de LEDs; por sua vez, os catodos dos LEDs estão conectados às colunas. O sistema funciona da seguinte maneira: as informações referentes à primeira linha da matriz são preenchidas; todas as colunas conterão as informações referentes a cada LED e, então, a linha é alimentada, ligando-se os LEDs correspondentes. O processo é repetido para cada linha, num processo de varredura. Como o processo é feito rapidamente, a mensagem na matriz parecerá estática aos olhos humanos.

³² Existem matrizes comerciais onde os LEDs são encapsulados em um único bloco. Os formatos usuais são: 7×5 , 8×8 e 16×16 .

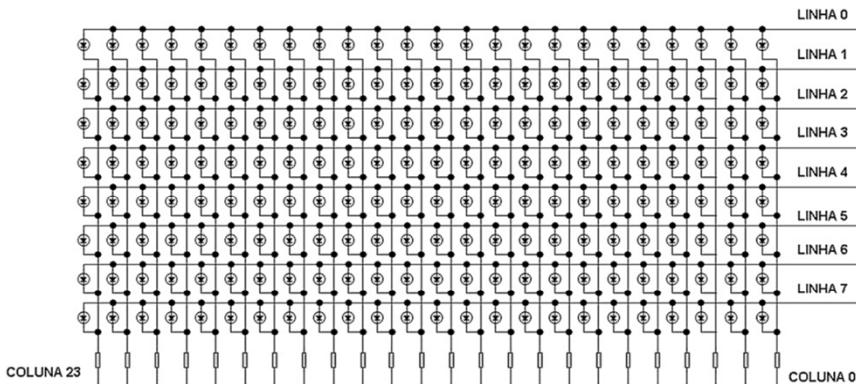


Fig. 11.13 – Matriz de LEDs com 8 linhas e 24 colunas.

Um circuito simplificado de controle para os 192 LEDs (24×8) da figura anterior é apresentado na fig. 11.14. Todos os LEDs são controlados com apenas 3 sinais, empregando-se a conversão serial-paralelo com o registrador de deslocamento 74HC595. Cada linha é alimentada com o emprego de um transistor PNP. Existe a necessidade de se empregar apenas um resistor por coluna, pois, na varredura, apenas um LED da coluna pode estar ligado. O programa de controle envia 24 bits de dados, correspondentes aos LEDs das colunas, mais 8 bits correspondentes à chave transistorizada a ser ligada (linha). Após esses 32 bits, o sinal de *strobe* habilita todos os LEDs da linha correspondente. Com oito linhas, serão oito trocas de linha na varredura.

No projeto do circuito, o transistor de cada linha deve ser capaz de suprir a corrente máxima dada pelo número de LEDs da linha multiplicado pela corrente máxima individual por LED. No caso acima, o 74HC595 pode drenar em torno de 8 mA por pino. Desta forma, se todos os LEDs de uma linha forem ligados, a corrente total da linha será de 192 mA (24×8 mA). Como na varredura das linhas (supondo 8), cada linha ficará ligada durante um período e por sete desligada, para a obtenção do maior brilho, é importante trabalhar com os LEDs na máxima corrente possível. Da mesma forma, o componente eletrônico responsável pela informação da

coluna (registrador de deslocamento) deve ser capaz de drenar a corrente exigida pelo LED da coluna. Caso ele não suporte a corrente desejada, um *driver* de corrente adequado³³ deve ser empregado (ver o apêndice D).

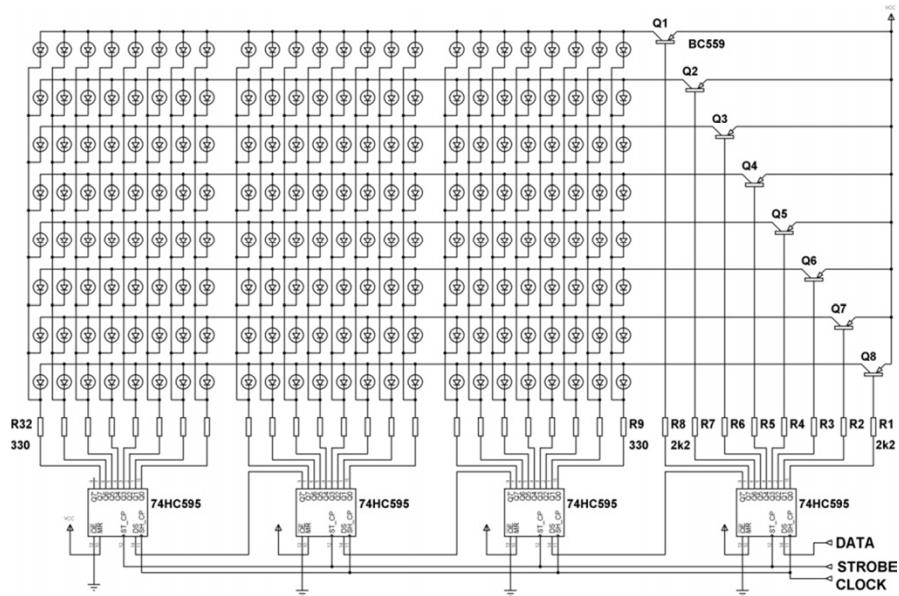


Fig. 11.14 – Circuito simplificado para o controle de uma matriz de 8×24 LEDs.

Na confecção da placa de circuito impresso, o desenhista colocará os LEDs da matriz na posição adequada de acordo com o desenho do circuito, ou utilizará matrizes comerciais. Na fig. 11.15, é ilustrada a visão tridimensional de uma placa de circuito impressa obtida no programa ARES (Proteus) para um circuito com 256 LEDs (8×32).

³³ O 74HC595 suporta em torno de 8 mA por pino, caso se deseje uma corrente maior, é necessário o emprego de um *driver* de corrente, como por exemplo o ULN2803. Existem também registradores de deslocamento com capacidade bem maior de corrente, como o TPIC6B595.

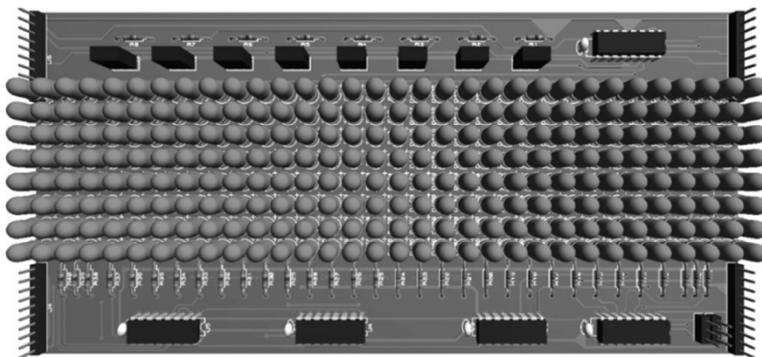


Fig. 11.15 – Visão tridimensional de uma matriz de LEDs .

Na programação de uma matriz de LEDs, a informação é colocada em uma tabela na memória do sistema de controle, que se encarrega de ler a informação, preencher adequadamente as informações dos registradores de deslocamento e gerar os sinais de controle. Em sistemas mais elaborados, como os grandes painéis de publicidade, um computador é o responsável pelo controle, com um software adequado convertendo as imagens para a matriz de LEDs do painel. Em sistemas complexos, que permitem a reprodução de várias cores, são empregados LEDs tricolores (*RGB – Red-Green-Blue*), tornando mais complexos o circuito e o software de controle.

11.7 CUBO DE LEDS

Um cubo de LEDs³⁴ é um conjunto de LEDs organizado de forma tridimensional. É empregado para animações gráficas, proporcionando um efeito visual interessante. O espaçamento entre os LEDs é tal, que a maioria dos LEDs da estrutura podem ser vistos (fig. 11.16).

³⁴ Cubos comerciais podem ser encontrados em www.seekway.com.cn.

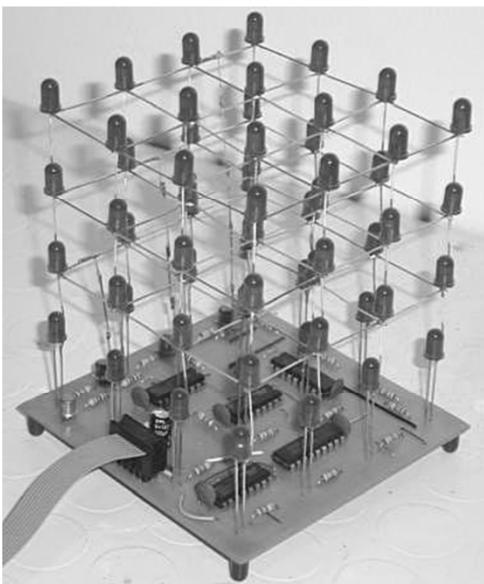


Fig. 11.16 – Cubo de LEDs de $4 \times 4 \times 4$ (64).

A mesma técnica empregada para o acionamento de uma matriz de LEDs é utilizada para o cubo (ver a seção anterior). Em uma matriz, os LEDs são dispostos em linhas e colunas, formando um plano bidimensional, onde todos os LEDs de uma linha são habilitados ao mesmo tempo, formando a imagem por varredura. Em um cubo, os LEDs devem ser organizados em planos horizontais sobrepostos, formando um arranjo tridimensional, conforme apresentado na fig. 11.16. Nela, cada plano é composto por 16 LEDs (4×4) e, ao todo, são 4 planos totalizando 64 LEDs (4×16).

Para a formação de uma imagem no cubo, pode ser empregada a varredura por plano. Ou seja, um plano horizontal inteiro de LEDs é acionado ao mesmo tempo. Assim, o acionamento sequencial dos planos forma uma imagem tridimensional. Em resumo, numa matriz de LEDs, uma linha inteira é energizada ao mesmo tempo na varredura, no cubo, um plano. Na geração dos dados para o cubo, é necessário que o

programador consiga visualizar tridimensionalmente a animação que deseja realizar e organizar adequadamente a varredura de acordo com o circuito de acionamento.

Na fig. 11.17, é apresentado o circuito eletrônico do cubo da fig. 11.16, pronto para ser conectado a um microcontrolador. O circuito emprega o CI ULN2803 para drenar a corrente dos LEDs (ver a seção D.2 do apêndice). Em cada plano horizontal, os anodos dos LEDs estão conectados, sendo a alimentação realizada através de uma chave transistorizada. Na sobreposição dos planos, onde os LEDs estarão uns sobre os outros, verticalmente, os catodos estão conectados e ligados às saídas dos CIs 4094³⁵ (responsáveis pela informação que irá aparecer em cada LED). Por exemplo, no circuito da fig. 11.17, os LEDs 1a, 2a, 3a e 4a estarão alinhados verticalmente.

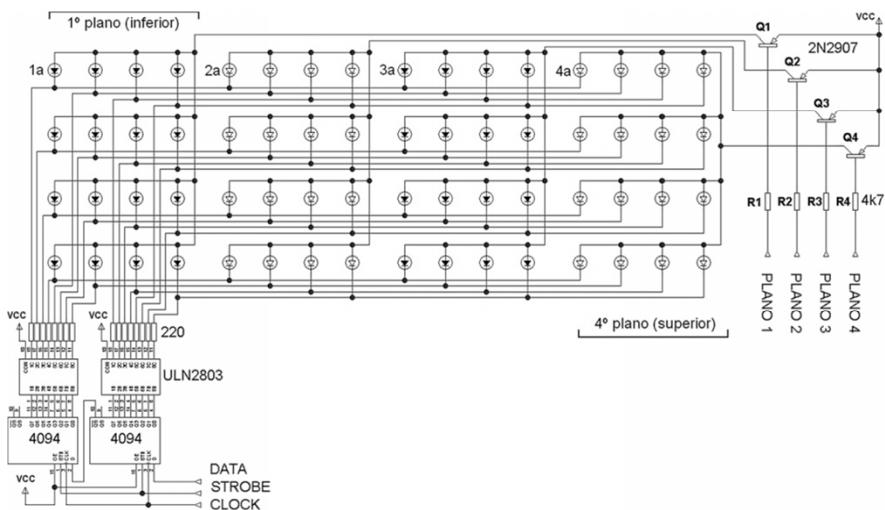


Fig. 11.17 – Circuito de um cubo de LEDs de $4 \times 4 \times 4$.

³⁵ Outro registrador de deslocamento similar ao 74HC595, entretanto com menor capacidade para suprir ou fornecer corrente.

A grande dificuldade no projeto de um cubo de LEDs é a montagem da estrutura tridimensional; quanto maior o número de LEDs, maior é a complexidade. Uma possibilidade é montar os planos horizontais individualmente com o emprego de um gabarito, e depois, um a um, soldá-los para a sobreposição. Na fig. 11.16, o plano inferior foi soldado diretamente na placa de circuito impresso, os demais sobrepostos a ele, através da solda dos catodos dos LEDs.

Da mesma forma que em uma matriz, é possível empregar LEDs tricolores no cubo (RGB). Entretanto, a montagem será bem mais difícil e o circuito de controle bem mais complexo.

Exercícios:

11.5 – Na fig. 11.18, é apresentado um sistema para o controle de uma matriz com 192 LEDs (8×24). O sistema alimenta uma linha por vez. Os dados correspondentes a cada coluna, incluindo qual linha será alimentada, são fornecidos por um conjunto de registradores de deslocamento (*shift register* – 74HC594, conversor serial-paralelo). Para controlar todo o sistema são empregadas 3 saídas do microcontrolador.

- Faça um programa para apresentar uma mensagem estática na matriz de LEDs. Antes da programação pesquise os módulos de matriz de LEDs disponíveis no mercado.
- Como o programa pode apresentar mensagens em movimento? Neste caso o pino de limpeza dos registradores (MR) teria um papel importante?

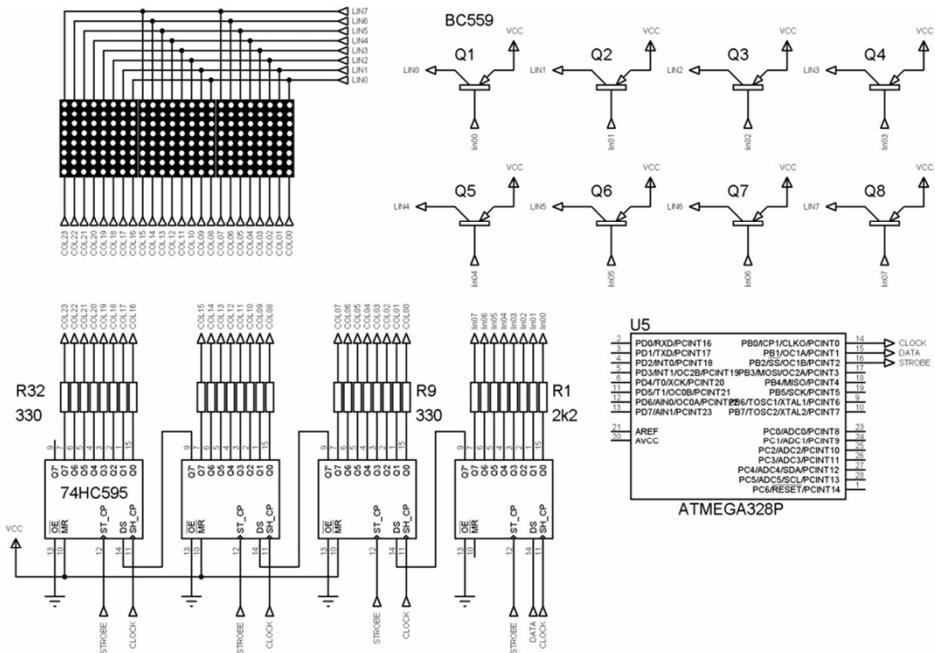


Fig. 11.18 – Matriz de LEDs e o 74HC595.

11.6 – Outra possibilidade para a montagem de um cubo de LEDs é ligar fisicamente os LEDs em colunas independentes, de forma a não existir ligação física entre as colunas, não sendo realizadas conexões na forma de planos. Como seria o circuito para o controle de tal cubo?

11.7 – Baseado no processo de multiplexação, empregando o ATmega328, projete um CLP (Controlador Lógico Programável) básico com 64 saídas e 64 entradas digitais.

12. DISPLAY GRÁFICO (128 × 64 pontos)

Antigamente, os projetos microcontrolados se restringiam ao uso de LCDs alfanuméricicos. Com o avanço tecnológico, os LCDs gráficos se tornaram comuns e as aplicações que os empregam cresceram rapidamente. Assim, o objetivo deste capítulo é introduzir um modelo comum de *display* gráfico, permitindo a elaboração de projetos com interfaces gráficas mais elaboradas. Também é apresentado um programa para a criação das imagens utilizadas na programação do LCD.

Um *display* gráfico é uma matriz de pontos visíveis pela aplicação de uma tensão elétrica sobre o cristal líquido de cada um desses pontos (pixel). Esse tipo de LCD é empregado para representar os mais diversos tipos de caracteres e figuras, cuja definição dependerá da quantidade de pontos do LCD.

Um tipo de LCD gráfico muito utilizado com microcontroladores é o de 128×64 pontos, baseado nos CIs controladores KS0108B e KS0107 ou similares. Os pontos não são ligados todos ao mesmo tempo, sendo a varredura do *display* realizada automaticamente pelo circuito de controle. Na fig. 12.1, o diagrama do circuito de controle com esses CIs é apresentado. Alguns modelos podem necessitar de uma fonte de tensão negativa para o contraste do LCD (fig. 12.1b). Entretanto, muitos modelos já possuem internamente o circuito que gera essa tensão (fig. 12.1a).

A descrição dos pinos de um LCD gráfico de 128×64 pontos é apresentada na tab. 12.1. Dependendo do fabricante, os pinos CS1 e CS2 podem ser ativos com diferentes níveis lógicos.

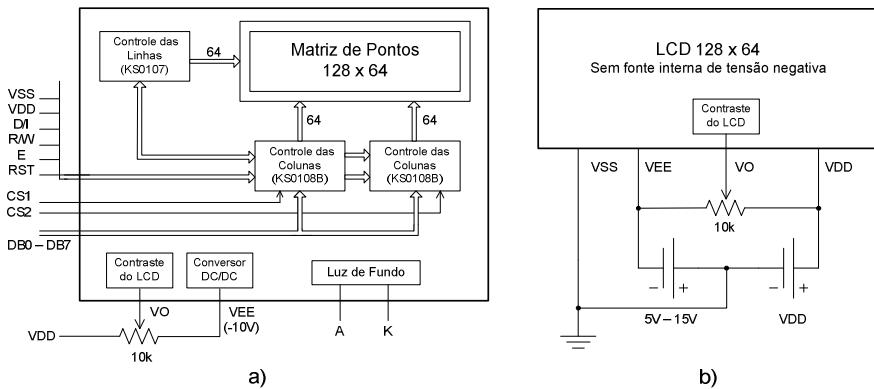


Fig. 12.1 – Diagrama esquemático de um LCD de 128×64 : a) com geração da tensão negativa para contraste e b) diagrama de ligação para um LCD sem fonte interna de tensão negativa.

Tab. 12.1 – Pinagem de um LCD gráfico de 128×64 pontos (obs.: a ordem dos pinos depende do fabricante).

Pino	Nome	Descrição
1	VSS	Terra.
2	VDD	Tensão de alimentação.
3	VO	Tensão para o contraste do LCD.
4	D/I	Sinal de Dados/Instrução.
5	R/W	Sinal de Leitura/Escrita.
6	E	Sinal de habilitação.
7	DB0	Via de dados [0 - 7] Permitem três estados (0, 1 e alta impedância).
8	DB1	
9	DB2	
10	DB3	
11	DB4	
12	DB5	
13	DB6	
14	DB7	
15	CS1	Seleção da primeira metade do LCD.
16	CS2	Seleção da segunda metade do LCD.
17	RST	Sinal de Reset.
18	VEE	Tensão negativa para o LCD.
19	A	Anodo do LED da iluminação de fundo.
20	K	Catodo do LED da iluminação de fundo.

O LCD gráfico possui uma memória RAM, na qual basta escrever a informação que se deseja apresentar, sendo a metade do *display* controlada por um CI KS0108B e a outra metade por outro. As posições de escrita seguem o padrão apresentado na fig. 12.2. As metades do LCD são separadas em 8 páginas, sendo cada página composta por 64 bytes. Cada metade do LCD é controlada individualmente pelos pinos CS1 e CS2. As colunas são numeradas da esquerda para a direita de 0 até 63. Uma imagem completa terá 1 kbyte de tamanho, $(128 \text{ bits} \times 64 \text{ bits})/8 = 1024 \text{ bytes}$. A escrita no LCD é feita por bytes e não se pode ligar pontos individuais diretamente.

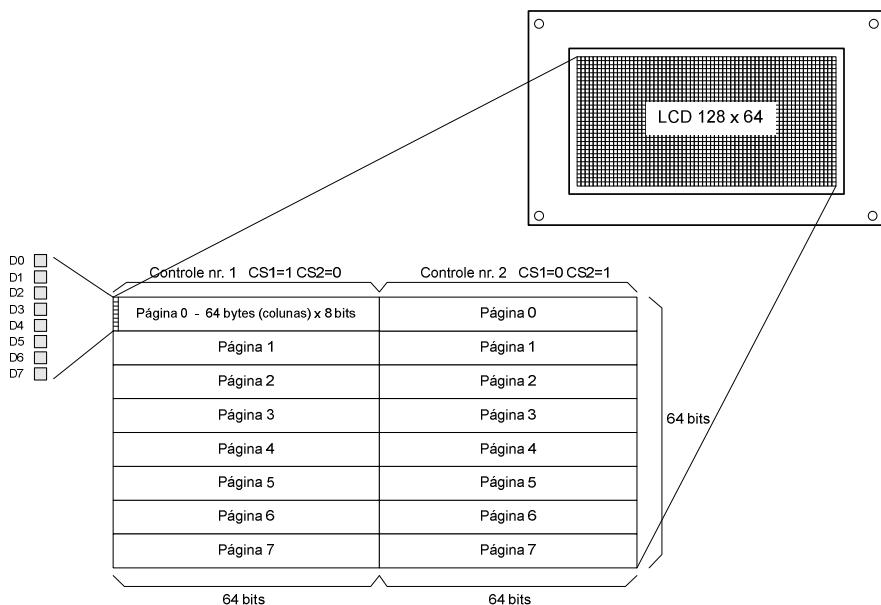


Fig. 12.2 – Organização da memória de pontos do LCD gráfico.

O correto funcionamento do LCD depende da inicialização do circuito de controle. O *reset* pode ser realizado externamente por uma rede RC ou controlado diretamente pelo microcontrolador. Os dados são transferidos após um pulso de habilitação (*enable*). Além disso, os tempos de resposta de cada modelo de LCD devem ser consultados no manual do fabricante. As instruções de controle do LCD são apresentadas na tab. 12.2.

Tab. 12.2 – Instruções de controle para o CI KS0108B.

Instrução	D/I	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Função
Liga/Desliga LCD	0	0	0	0	1	1	1	1	1	0/1	Liga/Desliga display. O status interno e a RAM não são afetados. (0 = desliga, 1 = liga)
Ajusta endereço	0	0	0	1	Endereço Y (0 - 63)						Ajusta o endereço Y no contador de endereços Y.
Ajusta Página (endereço X)	0	0	1	0	1	1	1	Página (0 - 7)			Ajusta o endereço X no registrador de endereço X.
Início da linha do LCD	0	0	1	1	Linha de início do Display (0 - 63)						Indica que os dados da RAM devem ser mostrados no topo do LCD.
Status de pronto	0	1	Ocupado	0	Ligado/Desligado	Reset	0	0	0	0	Ocupado: 0 = pronto para receber informações 1 = em operação. Ligado = 1 / Desligado = 0 Reset : 0 = normal 1 = reset.
Escreve dado no LCD	1	0	Dado para a escrita						Escreve dado (DB0:7) na RAM de dados do LCD. Após a escrita, o endereço Y é incrementado automaticamente.		
Lê dado do LCD	1	1	Dado para a leitura						Lê dado (DB0:7) da RAM do LCD.		

Para o correto funcionamento do LCD gráfico, alguns pontos devem ser considerados:

1. O controle do LCD é realizado entre a habilitação e desabilitação do mesmo (pulso de *enable*). Se CS1 ou CS2 não estiverem ativos, as instruções de entrada ou saída de dados não podem ser executadas.
2. Registradores de entrada no LCD armazenam dados temporariamente até que esses sejam escritos na sua RAM. Quando CS1 ou CS2 estiverem ativos, R/W e D/I selecionam o registrador de entrada.
3. Registradores de saída armazenam temporariamente os dados da RAM. Quando CS1 ou CS2 estiverem ativos e for realizada uma operação de leitura, o bit de *status* de ocupado (*busy flag*) pode ser lido. Para ler o conteúdo da RAM, são necessárias duas operações de leitura, uma para colocar o dado no registrador de saída e outro para lê-lo.
4. O sistema pode ser inicializado colocando o pino de *reset* em nível baixo durante a energização do circuito (pode ser feito por software). No *reset*, o *display* é desligado e é ajustada a linha de início de escrita para a posição zero. Somente a instrução de leitura pode ser realizada (teste do bit de ocupado e de *reset*, DB7 e DB4, respectivamente).

- 5.** O bit de ocupado indica se o KS0108B está pronto para uma leitura ou escrita. Quando estiver em nível alto, o KS0108B está realizando alguma operação interna; em nível baixo, aceita dados ou instruções. Em escritas rápidas do LCD, esse *flag* precisa ser monitorado, caso contrário, apenas se emprega um atraso adequado na escrita de um novo dado.
- 6.** É possível desligar a alimentação do cristal líquido (apagar o LCD), sem alterar o conteúdo da RAM.
- 7.** O registrador de página X indica qual página de dados da RAM será acessada. Cada troca de página precisa ser realizada por software.
- 8.** O contador de endereço Y indica qual coluna será escrita. É incrementado automaticamente após cada escrita ou leitura, da esquerda para a direita. Detalhe: o posicionamento de escrita não segue o sistema cartesiano, aqui o Y representa o *x* do sistema cartesiano e a página X representa o *y*.
- 9.** A memória RAM do *display* indica o estado de cada ponto da matriz de pontos. Um bit igual a 1 indica que o ponto estará ligado; para se apagar o ponto, o bit correspondente deve ser zerado.
- 10.** O registrador de linha de início indica que os dados da memória RAM devem ser apresentados na linha superior do LCD. É empregado quando se deseja deslocar alguma figura do LCD.

Na fig. 12.3, é indicado o processo de escrita do dado 0xAB (0b10101011) em uma coluna do LCD. Primeiramente, o LCD deve ser inicializado, habilita-se o pino CS1 ou CS2 e escolhe-se a coluna de escrita e a página. Após o dado ser colocado no barramento, avisa-se que será realizada uma operação de escrita e que o valor para o barramento é um dado. Então, aplica-se um pulso de habilitação (*enable*) para a transferência dos dados.

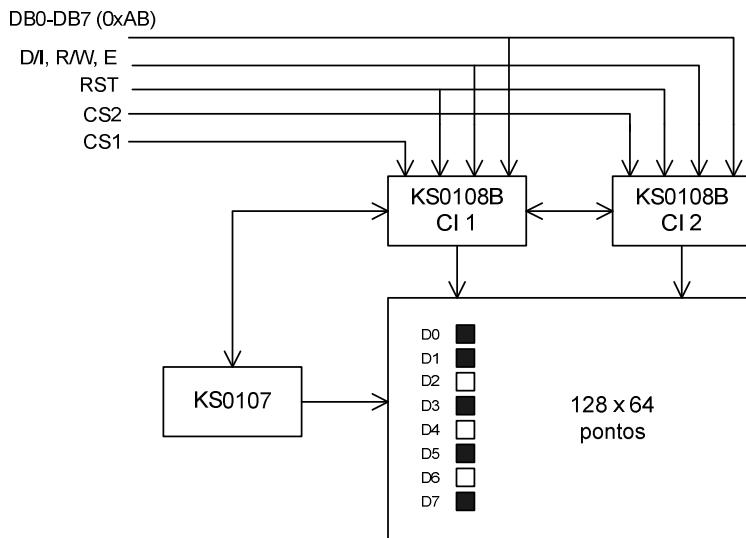


Fig. 12.3 – Escrita do dado 0xAB no LCD (fora de escala).

Na fig. 12.4, é apresentada a alteração de um único ponto do LCD. Para tal, primeiro é necessário uma leitura do byte da posição que se deseja alterar, salvando-se o seu conteúdo. Após, muda-se somente o bit desejado e reescreve-se o novo byte na posição lida. No exemplo da fig. 12.4, o bit D2 foi alterado.

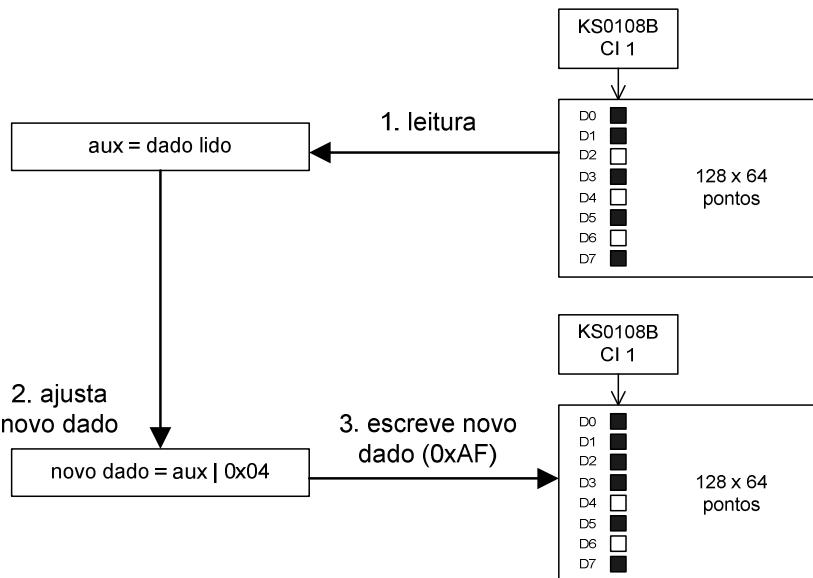


Fig. 12.4 – Escrevendo em um único ponto do LCD.

12.1 CONVERSÃO DE FIGURAS

Programas gratuitos permitem converter figuras em tabela de dados que podem ser acessados diretamente pelo programa para o envio ao *display* gráfico. O único trabalho é adequar a tabela criada com a forma de escrita da memória RAM do LCD.

Para criar uma figura, por exemplo, pode-se empregar o *Bitmap2LCD* (versão gratuita disponível em www.bitmap2lcd.com). Esse programa é utilizado para gerar tabelas de dados para inúmeros *displays* gráficos, basta desenhar ou carregar um desenho pronto. Quando o programa é inicializado são abertas 3 janelas: principal, desenho e seleção de arquivo; elas podem ser vistas na fig. 12.5. Uma das primeiras configurações é o ajuste do número de pontos do LCD, no menu <LCD_matrix> da janela principal.

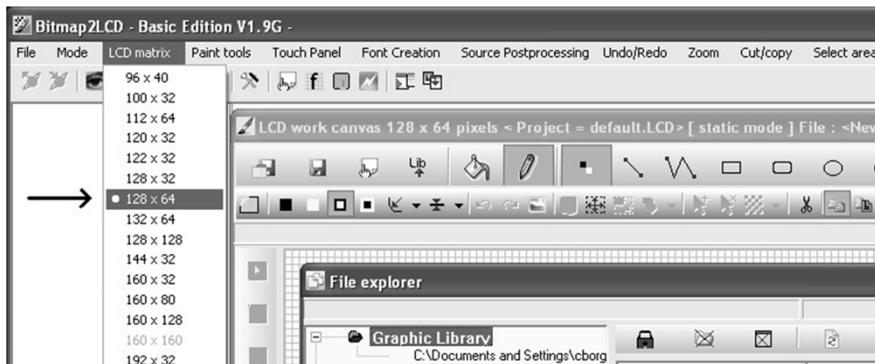


Fig. 12.5 – Configurando o LCD empregado.

Após a escolha do modelo de LCD, pode-se desenhar ou carregar um desenho já confeccionado. Por exemplo, é fácil empregar o *Paint* (Windows) para gerar uma figura com o número de pixels adequados. A figura pode ser selecionada na janela de seleção de arquivo, como exemplificado na fig. 12.6.

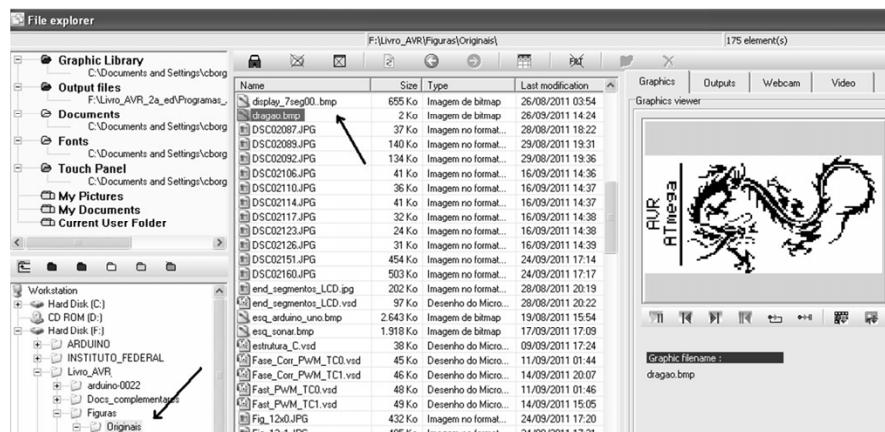


Fig. 12.6 – Carregando uma figura pronta.

Com o desenho carregado, ajustam-se as configurações para a criação da tabela com os dados do desenho (fig. 12.7). Ao se clicar no menu <File> <Configuration and Preferences> da janela principal, será aberta a janela de configuração da fig. 12.8, na qual se ajustam o formato dos dados e sua

forma de escrita para serem salvos em um arquivo *.h (próprio para ser carregado em um programa em C).

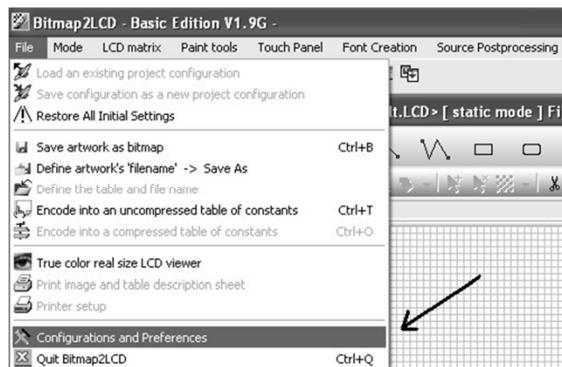


Fig. 12.7 – Configurando as preferências para a geração da tabela.

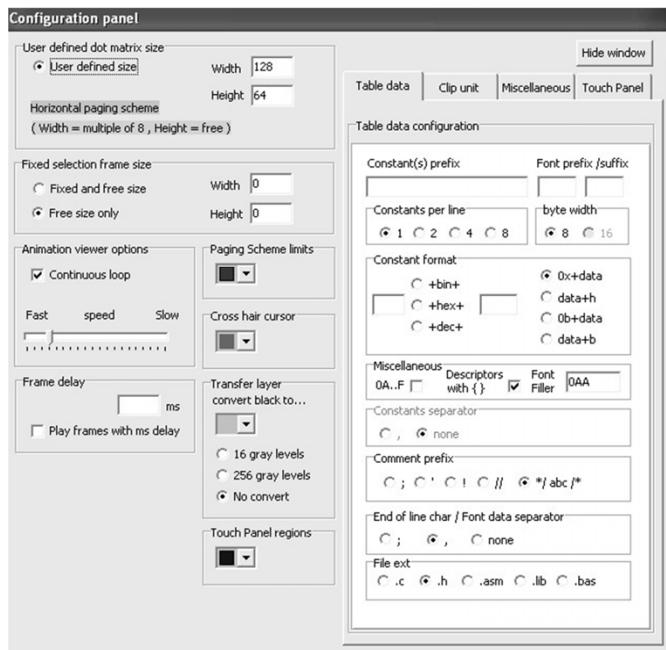


Fig. 12.8 – Janela para ajuste das preferências da tabela.

Após a configuração da forma de dados no arquivo de saída, ainda falta definir como o programa deverá interpretar o desenho e codificar os dados de saída. Na janela de seleção de arquivo, onde aparece a figura aberta, na aba <Outputs> (figs. 12.9-10) devem ser selecionados o ícone para indicar que os dados serão gravados por bytes com codificação vertical; o ícone para indicar que o bit mais significativo será escrito por último e o ícone para indicar o ponto de origem para a codificação (superior esquerdo). Essas configurações são adequadas para o LCD empregado neste capítulo.

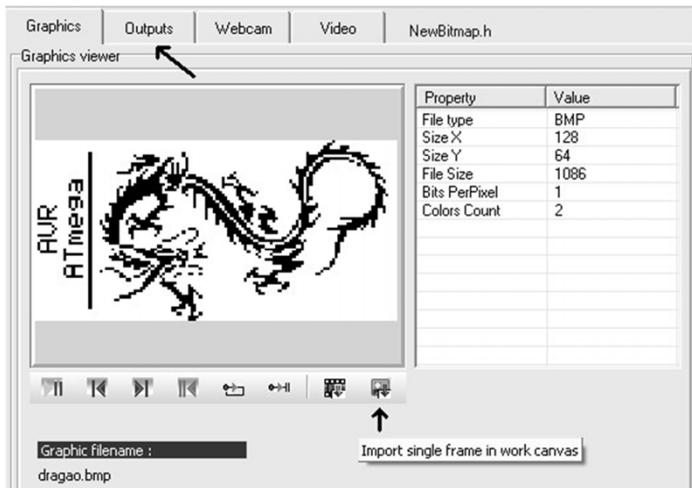


Fig. 12.9 – Aba <Outputs> para configuração dos dados e ícone para importar a figura para a janela de desenho.

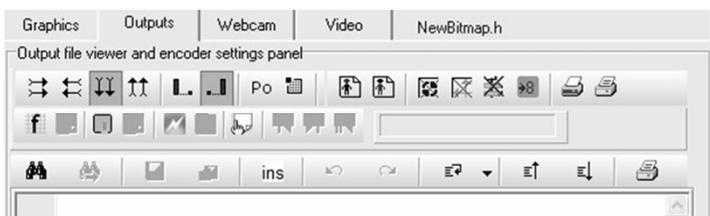


Fig. 12.10 – Ícones para a configuração dos dados.

Após o ajuste de todas as configurações, basta clicar no ícone <Import single frame in work canvas> da janela de seleção de arquivo (fig. 12.9) e a imagem é carregada para a área da janela de desenho (fig. 12.11). Agora com essa janela ativa é necessário o comando de colar <Ctrl + V>. Isso parece estranho, mas somente apóes esse passo é que a figura pode ser codificada na tabela de dados. Para isso clica-se no ícone  <Encode artwork into table of constants>. Feito isso, o programa irá pedir o nome e o endereço para salvamento do arquivo de dados gerados e, na tela da seleção de arquivo, ao lado da figura, irão aparecer os dados (fig. 12.12). Com os dados gerados, basta adequá-los para uso no programa desenvolvido.

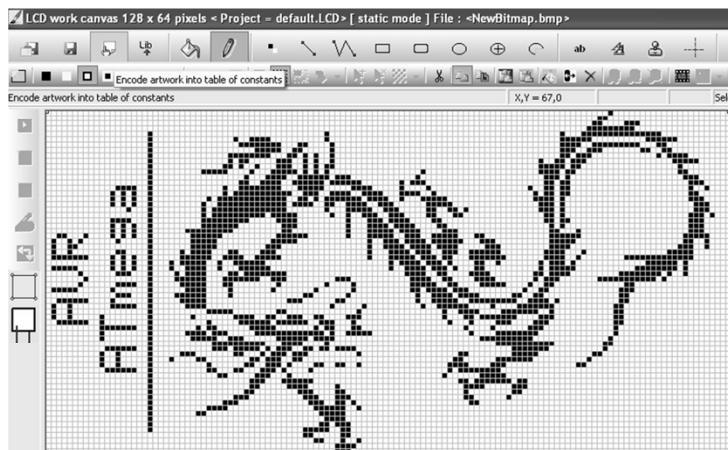


Fig. 12.11 – Janela de desenho para a codificação dos dados.

Fig. 12.12 - Resultado da codificação dos dados.

A seguir, são apresentados os arquivos de programa para mostrar a imagem da área de desenho da fig. 12.11 em um LCD 128 × 64 empregando uma ATmega328. O circuito pode ser visto na fig. 12.13.

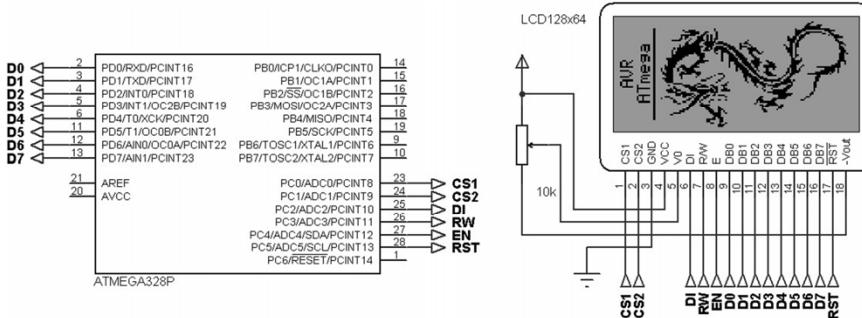


Fig. 12.13 – Circuito para o controle de um LCD 128 × 64.

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL //define a frequência do microcontrolador - 16MHz

#include <avr/io.h> //definições do componente especificado
#include <util/delay.h> //biblioteca para o uso das rotinas de _delay_ms e _delay_us()
#include <avr/pgmspace.h> //para o uso do PROGMEM, gravação de dados na memória flash

//Definições de macros para o trabalho com bits
#define set_bit(y,bit) (y|=(1<<bit))//coloca em 1 o bit x da variável Y
#define clr_bit(y,bit) (y&=~(1<<bit))//coloca em 0 o bit x da variável Y
#define cpl_bit(y,bit) (y^=(1<<bit))//troca o estado lógico do bit x da variável Y
#define tst_bit(y,bit) (y&(1<<bit)) //retorna 0 ou 1 conforme leitura do bit

#endif
```

LCDG_128x64.c (programa principal)

```
#include "def_principais.h"
#include "LCDG.h"
#include "dragao.h" //figura com 1024 bytes

int main(void)
{
    DDRD = 0xFF; //pinos do PORTD como saída
    DDRC = 0xFF; //pinos do PORTC como saída
    UCSR0B= 0x00; //para uso dos pinos do PORTD no Arduino

    inic_LCDG();
    escreve_todo_LCDG(dragao);
    while(1{}) //código principal
}
```

LCDG.h (arquivo de cabeçalho do LCDG.c)

```
#ifndef _LCDG_H
#define _LCDG_H

#include "def_principais.h"

#define LIGA_LCD 0x3F
#define DESL_LCD 0x3E
#define LIN_INIC 0xC0
#define Y_INIC 0x40
#define PAG_INIC 0xB8

#define DADOS PORTD //Px0:7 nos pinos de dados DB0:DB7
#define CS1 PC0 //ativo em alto
#define CS2 PC1 //ativo em alto
#define DI PC2 //DI=1 dados, DI=0 instrução
#define RW PC3 //RW=0 escrita, RW=1 leitura
#define EN PC4 //ativo em alto
#define RST PC5 //ativo em baixo

#define set_escrita() _delay_us(1); clr_bit(PORTC,RW)
#define set_leitura() _delay_us(1); set_bit(PORTC,RW)
#define set_dado() _delay_us(1); set_bit(PORTC,DI)
#define set_instrucao() _delay_us(1); clr_bit(PORTC,DI)
```

```

#define set_CS1()          _delay_us(1);  set_bit(PORTC,CS1)
#define set_CS2()          _delay_us(1);  set_bit(PORTC,CS2)
#define clr_CS1()          _delay_us(1);  clr_bit(PORTC,CS1)
#define clr_CS2()          _delay_us(1);  clr_bit(PORTC,CS2)
#define set_enable()        _delay_us(5);   set_bit(PORTC,EN)
#define clr_enable()        _delay_us(5);   clr_bit(PORTC,EN)
#define set_reset()         _delay_us(10);  set_bit(PORTC,RST)
#define clr_reset()         _delay_us(10);  clr_bit(PORTC,RST)

#define reset_LCDG()       clr_reset();   set_reset()
#define enable_LCDG()      set_enable();  clr_enable()

void inic_LCDG();
void escreve_LCDG(unsigned char dado, unsigned char coluna, unsigned char pagina);
void limpa_LCDG(unsigned char limpa);
void escreve_caractere_LCDG(unsigned char caractere, unsigned char col, unsigned char pag);
void escreve_stringFLASH_LCDG(char *c, unsigned char col, unsigned char pag);
void escreve_stringRAM_LCDG(char *c, unsigned char col, unsigned char pag);
void escreve_todo_LCDG(unsigned char *figura);

#endif

```

LCDG.c (arquivo com as funções para trabalho com o LCD gráfico)

```

#include "LCDG.h"
#include "tabela_ASCII.h"

//-----
//Inicialização padrão, as inicializações devem ser feitas conforme necessário.
//-----
void inic_LCDG()
{
    set_CS1();
    set_CS2();
    reset_LCDG();
    set_instrucao();
    set_escrita();
    DADOS = LIGA_LCD;
    enable_LCDG();
    DADOS = LIN_INIC;
    enable_LCDG();
    DADOS = Y_INIC;
    enable_LCDG();
    DADOS = PAG_INIC;
    enable_LCDG();

    limpa_LCDG(0x00);
}
//-----
//Escreve 1 byte na coluna e pagina especificada
//-----
void escreve_LCDG(unsigned char dado, unsigned char coluna, unsigned char pagina)
{
    set_instrucao();//instrução
    if (coluna>63)//coluna 0-127
    {
        clr_CS1();
        set_CS2();
        DADOS = Y_INIC + coluna - 64;
    }
}

```

```

    else
    {
        clr_CS2();
        set_CS1();
        DADOS = Y_INIC + coluna;
    }

    enable_LCDG();
    DADOS = PAG_INIC + pagina;      //página 0 - 7
    enable_LCDG();

    //escreve dado
    set_dado();
    DADOS = dado;
    enable_LCDG();
}

//-----
//Limpa o LCDG - 0x00 apaga todos os pixels e 0xFF liga
//-----
void limpa_LCDG(unsigned char limpa)
{
    unsigned char pag, col;

    for(pag=0;pag<8;pag++)
    {
        for(col=0;col<128;col++)
            escreve_LCDG(limpa, col, pag);
    }
}

//-----
//Escreve uma imagem completa (1024 bytes)
//-----
void escreve_todo_LCDG(unsigned char *figura)
{
    unsigned char pag, col;

    for(pag=0;pag<8;pag++)
    {
        for(col=0;col<128;col++)
        {
            escreve_LCDG(pgm_read_byte(&(*figura)), col, pag);
            figura++;
        }
    }
}

//-----
//Escreve um caractere da tabela ASCII (igual ao LCD com controlador HD44780)
//-----
void escreve_caractere_LCDG(unsigned char caractere, unsigned char col, unsigned char pag)
{
    unsigned char i;

    for (i=0; i<5; i++)
        escreve_LCDG(pgm_read_byte(&tabela_ASCII[caractere-32][i]), col+i, pag);

    escreve_LCDG(0, col+5, pag); //uma coluna em branco após o caractere
}

```

```

//-----
//Escreve um conjunto de caracteres gravados na FLAHS, conforme tabela ASCII
//-----
void escreve_stringFLASH_LCDG(char *c, unsigned char col, unsigned char pag)
{
    unsigned char j=0;

    for(;pgm_read_byte(&(*c))!=0;c++)
    {
        escreve_caractere_LCDG(pgm_read_byte(&(*c)), col+6*j, pag);
        j++;
    }
}
//-----
//Escreve um conjunto de caracteres gravados na RAM, conforme tabela ASCII
//-----
void escreve_stringRAM_LCDG(char *c, unsigned char col, unsigned char pag)
{
    unsigned char j=0;
    for(;*c!=0;c++)
    {
        escreve_caractere_LCDG(*c, col+6*j, pag);
        j++;
    }
}
//-----

```

dragao.h (apenas um trecho da tabela com a figura, são 1024 bytes de dados)

```

prog_uchar dragao[1024] = {
    0x00 /* _____ */,
    0x00 /* _____ */,
    ...
    0xF0 /* #####_____ */,
    0x00 /* _____ */,
    0xE0 /* ####_____ */,
    0xE0 /* ####_____ */,
    ...
    0x00 /* _____ */,
    0x00 /* _____ */
};
```

As funções apresentadas não fazem a leitura do bit de ocupado do LCD (*busy flag*). Dependendo do LCD utilizado o tempo de resposta pode ser outro e as rotinas de atraso podem necessitar de tempos maiores, os quais podem ser corrigidos no arquivo **LCDG.h**.

12.2 USANDO O DISPLAY GRÁFICO COMO UM LCD 21 x 8

Além de imagens, o LCD gráfico é muito utilizado para a escrita de mensagens alfanuméricas. Assim, é importante dispor de funções que facilitem a sua escrita. Desta forma, foram desenvolvidas funções para escrever no *display* gráfico como se ele fosse um LCD alfanumérico comum (controlador HD44780). Na fig. 12.14, é apresentada a figura com os caracteres imprimíveis da tabela ASCII (código 32 até o 126) iguais ao LCD alfanumérico. Cada caractere é formado por uma matriz de 5×8 pixels, sendo consumidos 5 bytes de informação por caractere e sendo o bit mais significativo zero para garantir o espaçamento entre as linhas. Então, como apresentado na seção anterior, os caracteres foram codificados nos seus bytes individuais com cada pixel ativo representando o número 1 binário. Na tab. 12.3, é apresentado o resultado dessa codificação, já no formato de matriz para uso na programação.

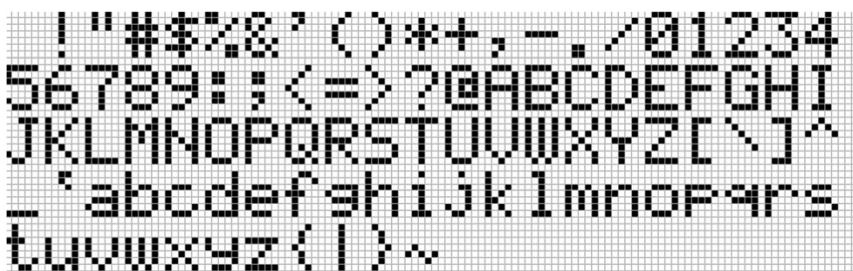


Fig. 12.14 – Desenho dos caracteres no formato ASCII para um LCD 128×64 .

Tab. 12.3 – Tabela com os dados para a escrita de caracteres ASCII num LCD 128×64 .

tabela_ASCII.h

```
//Caracteres ASCII, começando com o código 32 até o 126
prog_uchar tabela_ASCII[95][5] = {
{0x00, 0x00, 0x00, 0x00, 0x00}, // 0      {0x7F, 0x09, 0x09, 0x09, 0x06}, // P 48
{0x00, 0x00, 0x5F, 0x00, 0x00}, // ! 1    {0x3E, 0x41, 0x51, 0x21, 0x5E}, // Q 49
{0x00, 0x07, 0x00, 0x07, 0x00}, // " 2    {0x7F, 0x09, 0x19, 0x29, 0x46}, // R 50
{0x14, 0x7F, 0x14, 0x7F, 0x14}, // # 3   {0x46, 0x49, 0x49, 0x49, 0x31}, // S 51
{0x24, 0x2A, 0x7F, 0x2A, 0x12}, // $ 4   {0x01, 0x01, 0x7F, 0x01, 0x01}, // T 52
{0x23, 0x13, 0x08, 0x64, 0x62}, // % 5   {0x3F, 0x40, 0x40, 0x40, 0x3F}, // U 53
{0x36, 0x49, 0x55, 0x22, 0x50}, // & 6   {0x1F, 0x20, 0x40, 0x20, 0x1F}, // V 54
{0x00, 0x05, 0x03, 0x00, 0x00}, // ' 7   {0x3F, 0x40, 0x3F, 0x40, 0x3F}, // W 55
{0x00, 0x1C, 0x22, 0x41, 0x00}, // ( 8   {0x63, 0x14, 0x08, 0x14, 0x63}, // X 56
```

```

{0x00, 0x41, 0x22, 0x1C, 0x00}, // ) 9
{0x14, 0x08, 0x3E, 0x08, 0x14}, // * 10
{0x08, 0x08, 0x3E, 0x08, 0x08}, // + 11
{0x00, 0x50, 0x30, 0x00, 0x00}, // , 12
{0x08, 0x08, 0x08, 0x08, 0x08}, // - 13
{0x00, 0x60, 0x60, 0x00, 0x00}, // . 14
{0x20, 0x10, 0x08, 0x04, 0x02}, // / 15
{0x3E, 0x51, 0x49, 0x45, 0x3E}, // \ 16
{0x00, 0x42, 0x7F, 0x40, 0x00}, // ] 17
{0x42, 0x61, 0x51, 0x49, 0x46}, // ^ 18
{0x21, 0x41, 0x45, 0x4B, 0x31}, // _ 19
{0x18, 0x14, 0x12, 0x7F, 0x10}, // = 20
{0x27, 0x45, 0x45, 0x45, 0x39}, // ; 21
{0x3C, 0x4A, 0x49, 0x49, 0x30}, // : 22
{0x01, 0x01, 0x79, 0x05, 0x03}, // < 23
{0x36, 0x49, 0x49, 0x49, 0x36}, // / 24
{0x06, 0x49, 0x49, 0x29, 0x1E}, // ? 25
{0x00, 0x36, 0x36, 0x00, 0x00}, // : 26
{0x00, 0x56, 0x36, 0x00, 0x00}, // ; 27
{0x08, 0x14, 0x22, 0x41, 0x00}, // < 28
{0x14, 0x14, 0x14, 0x14, 0x14}, // = 29
{0x41, 0x22, 0x14, 0x08, 0x00}, // > 30
{0x02, 0x01, 0x51, 0x09, 0x06}, // ? 31
{0x3E, 0x41, 0x5D, 0x55, 0x5E}, // @ 32
{0x7E, 0x09, 0x09, 0x09, 0x7E}, // / A 33
{0x7F, 0x49, 0x49, 0x49, 0x36}, // B 34
{0x3E, 0x41, 0x41, 0x41, 0x22}, // C 35
{0x7F, 0x41, 0x41, 0x22, 0x1C}, // D 36
{0x7F, 0x49, 0x49, 0x41, 0x41}, // E 37
{0x7F, 0x09, 0x09, 0x09, 0x01}, // F 38
{0x3E, 0x41, 0x41, 0x49, 0x7A}, // G 39
{0x7F, 0x08, 0x08, 0x08, 0x7F}, // H 40
{0x00, 0x41, 0x7F, 0x41, 0x00}, // I 41
{0x20, 0x40, 0x41, 0x3F, 0x01}, // J 42
{0x7F, 0x08, 0x14, 0x22, 0x41}, // K 43
{0x7F, 0x40, 0x40, 0x40, 0x40}, // L 44
{0x7F, 0x02, 0x0C, 0x02, 0x7F}, // M 45
{0x7F, 0x04, 0x08, 0x10, 0x7F}, // N 46
{0x3E, 0x41, 0x41, 0x41, 0x3E}, // O 47
{0x07, 0x08, 0x78, 0x08, 0x07}, // Y 57
{0x61, 0x51, 0x49, 0x45, 0x43}, // Z 58
{0x00, 0x7F, 0x41, 0x41, 0x00}, // [ 59
{0x01, 0x02, 0x04, 0x08, 0x10}, // \ 60
{0x00, 0x41, 0x41, 0x7F, 0x00}, // ] 61
{0x04, 0x02, 0x01, 0x02, 0x04}, // ^ 62
{0x40, 0x40, 0x40, 0x40, 0x40}, // _ 63
{0x00, 0x00, 0x03, 0x05, 0x00}, // = 64
{0x20, 0x54, 0x54, 0x54, 0x78}, // a 65
{0x7F, 0x48, 0x44, 0x44, 0x38}, // b 66
{0x38, 0x44, 0x44, 0x44, 0x20}, // c 67
{0x38, 0x44, 0x44, 0x48, 0x7F}, // d 68
{0x38, 0x54, 0x54, 0x54, 0x18}, // e 69
{0x08, 0x7E, 0x09, 0x01, 0x02}, // f 70
{0x08, 0x54, 0x54, 0x54, 0x3C}, // g 71
{0x7F, 0x08, 0x04, 0x04, 0x78}, // h 72
{0x00, 0x44, 0x7D, 0x40, 0x00}, // i 74
{0x20, 0x40, 0x45, 0x3C, 0x00}, // j 75
{0x7F, 0x10, 0x28, 0x44, 0x00}, // k 76
{0x00, 0x41, 0x7F, 0x40, 0x00}, // l 77
{0x7C, 0x04, 0x38, 0x04, 0x78}, // m 78
{0x7C, 0x08, 0x04, 0x04, 0x78}, // n 79
{0x38, 0x44, 0x44, 0x44, 0x38}, // o 80
{0x7C, 0x14, 0x14, 0x14, 0x08}, // p 81
{0x08, 0x14, 0x14, 0x14, 0x18}, // q 82
{0x7C, 0x08, 0x04, 0x04, 0x08}, // r 83
{0x48, 0x54, 0x54, 0x54, 0x20}, // s 84
{0x04, 0x3F, 0x44, 0x40, 0x20}, // t 85
{0x3C, 0x40, 0x40, 0x20, 0x7C}, // u 86
{0x1C, 0x20, 0x40, 0x20, 0x1C}, // v 87
{0x3C, 0x40, 0x3C, 0x40, 0x3C}, // w 88
{0x44, 0x28, 0x10, 0x28, 0x44}, // x 89
{0x0C, 0x50, 0x50, 0x50, 0x3C}, // y 90
{0x44, 0x64, 0x54, 0x4C, 0x44}, // z 91
{0x00, 0x08, 0x36, 0x41, 0x00}, // { 92
{0x00, 0x00, 0x7F, 0x00, 0x00}, // | 93
{0x00, 0x41, 0x36, 0x08, 0x00}, // } 94
{0x10, 0x08, 0x10, 0x20, 0x10} // ~ 95
};


```

Como o LCD gráfico possui 64 pixels na vertical (8 bytes) e cada caractere possui 1 byte de altura, é possível utilizar 8 linhas para a escrita. Como cada caractere possui 5 pixels de largura e lembrando que pelo menos 1 pixel deve ser deixado para a escrita entre os caracteres, é possível escrever 21 caracteres por linha ($6 \times 21 = 126$ pixels), sobrando ainda 2 pixels. Desta forma, o LCD 128×64 pode ser escrito como se fosse um LCD alfanumérico de 21 caracteres por 8 linhas!

Os caracteres definidos para as funções de escrita no LCD gráfico estão no arquivo **LCDG.c** apresentado anteriormente. A seguir, o uso dessas funções é exemplificado. Para a escrita, basta passar para as

funções a mensagem, a coluna e a linha onde se deseja escrever. O resultado prático do programa é apresentado na fig. 12.15.

LCD_21x8.c

```
#include "def_principais.h"
#include "LCDG.h"

//teste da escrita com os caracteres ASCII
prog_char ascii0[] = " !\"#$%&'()*+,.-./01234\0";
prog_char ascii1[] = "56789:;<=>?@ABCDEFGHI\0";
prog_char ascii2[] = "JKLMNOPQRSTUVWXYZ[\\" ]^@\0";
prog_char ascii3[] = "_`abcdefghijklmnopqrstuvwxyz\0";
prog_char ascii4[] = "uvwxyz{|}~\0";

int main(void)
{
    DDRD = 0xFF;           //pinos do PORTD como saída
    DDRC = 0xFF;           //pinos do PORTC como saída
    UCSR0B= 0x00;          //para uso dos pinos do PORTD no Arduino

    inic_LCDG();

    escreve_stringFLASH_LCDG(ascii0,0,0);
    escreve_stringFLASH_LCDG(ascii1,0,1);
    escreve_stringFLASH_LCDG(ascii2,0,2);
    escreve_stringFLASH_LCDG(ascii3,0,3);
    escreve_stringFLASH_LCDG(ascii4,0,4);
    escreve_stringRAM_LCDG("LCD 128x64 COMO UM",10,6);
    escreve_stringRAM_LCDG("ALFANUMERICO 21x8",13,7);

    while(1{})           //código principal
}
```



Fig. 12.15 – Resultado prático para a escrita de caracteres num LCD 128 × 64.

Exercícios:

- 12.1** – Elaborar um programa para gerar uma animação gráfica com 5 quadros para o circuito da fig. 12.13.
- 12.2** – Alterar as funções apresentadas para o LCD 128×64 para que seja feita a leitura do seu bit de ocupado (*busy flag*).
- 12.3** – Alterar as funções apresentadas para a escrita de mensagens com pixels invertidos no LCD 128×64 , ou seja, escrita clara com fundo escuro.
- 12.4** – Crie uma função para a leitura dos dados diretamente do LCD 128×64 , permitindo a alteração de um único pixel se desejado.
-

13. FORMAS DE ONDA E SINAIS ANALÓGICOS

A geração de formas de onda por um sistema microcontrolado é amplamente utilizada em sistemas digitais que possuem uma interface analógica. A criação de sinais analógicos é fundamental para o projeto de sistemas que interagem com grandezas físicas. Dessa forma, alguns conceitos básicos necessários para a geração de formas de onda são apresentados neste capítulo, incluindo exemplos do emprego da modulação por largura de pulso (PWM) e técnicas para a conversão de um sinal digital para analógico.

13.1 DISCRETIZANDO UM SINAL

Para transformar um sinal analógico em digital, tanto o tempo quanto a amplitude devem ser discretizados, o que é denominado amostragem e quantização, respectivamente. Para a amostragem, define-se um período (intervalo de tempo entre cada amostra) e, consequentemente, uma frequência de amostragem (número de amostras por segundo). Para um sinal senoidal, por exemplo:

Sinal Contínuo:

$$y(t) = A \cdot \operatorname{sen}(2\pi \cdot f \cdot t) \quad [\text{s}] \quad (13.1)$$

Sinal Discreto:

$$Y[n] = A \cdot \operatorname{sen}(2\pi \cdot f \cdot n \cdot \Delta t), \quad (13.2)$$

onde A = amplitude do sinal, f = frequência do sinal, t = tempo, n = índice do vetor (inteiro positivo, 0, 1, ... N) e $\Delta t = 1/f_s$, (f_s é a frequência de amostragem do sinal - *sampling frequency*). A frequência de amostragem deve ser no mínimo duas vezes maior que a maior frequência presente no sinal a ser amostrado (Teorema de Nyquist).

Na fig. 13.1, é ilustrada a amostragem de um sinal senoidal representado por: $y(t) = 3 \cdot \operatorname{sen}(2\pi \cdot 1 \cdot t)$.

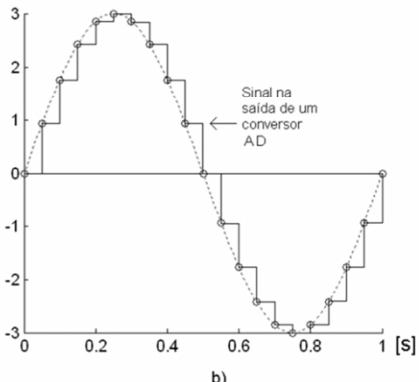
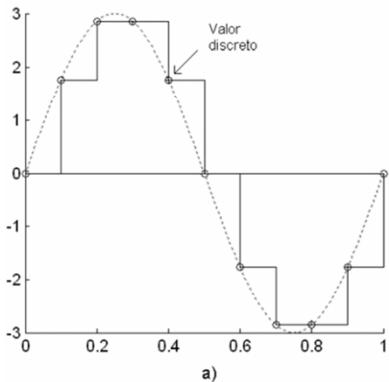


Fig. 13.1 – Amostragem de um sinal senoidal de 1 Hz: a) $f_s = 10$ Hz e b) $f_s = 20$ Hz.

Na fig. 13.1a, o sinal apresenta uma frequência de amostragem de 10 Hz (10 amostras por segundo), na fig. 13.1b, essa frequência é de 20 Hz. Considerando n começando em zero, tem-se, respectivamente, 11 e 21 amostras no intervalo de tempo de 1 s (um período de amostragem do sinal). Os pontos amostrados da fig. 13.1 são dados pela eq. 13.2, resultando, para a fig. 13.1a, em:

$$Y_a[n] = 3 \cdot \text{sen}(2\pi \cdot n \cdot 0,1),$$

com $n = 0, 1, 2, \dots, 10$. Assim:

$$Y_a = [0 \ 1,76 \ 2,85 \ 2,85 \ 1,76 \ 0 \ -1,76 \ -2,85 \ -2,85 \ -1,76 \ 0],$$

e para a fig. 13.1b:

$$Y_b[n] = 3 \cdot \text{sen}(2\pi \cdot n \cdot 0,05),$$

com $n = 0, 1, 2, \dots, 20$, o que resulta:

$$Y_b = [0 \ 0,93 \ 1,76 \ 2,43 \ 2,85 \ 3 \ 2,85 \ 2,43 \ 1,76 \ 0,93 \ 0 \ -0,93 \ -1,76 \ -2,43 \ -2,85 \ -3 \ -2,85 \ -2,43 \ -1,76 \ -0,93 \ 0]$$

Se for usado um conversor Analógico-Digital (AD), um valor é mantido constante na saída do mesmo até que um novo valor seja convertido (operação de amostragem e retenção – SH, *Sample and Hold*). O valor é quantizado de acordo com o número de bits de resolução do conversor AD.

Em sistemas microcontrolados, informações referentes aos pontos de uma forma onda podem ser armazenadas na memória. Enviando-se essas informações a um conversor Digital-Analógico (DA), numa taxa pré-definida e na sequência adequada, é possível a recomposição do sinal original. Como por exemplo, a leitura de um sinal de áudio por um MP3 *player*.

Exemplo de Cálculo dos valores para uma senóide

Considerando 8 bits de dados – sem sinal, uma senóide deve apresentar valores de 0 até 255 com nível médio no valor 127. O número de pontos que compõem a senóide deve ser escolhido de acordo com a precisão desejada na escala temporal e respeitando-se o Teorema de Nyquist.

No exemplo a seguir, foi calculada uma senóide com amplitude de 0 a 255 (8 bits) e 256 pontos (255 intervalos em 360° , resolução de $360/255^\circ$). O cálculo é simples e a senóide é computada por:

$$Valor_{seno}[n] = \text{sen}(n \cdot 360^\circ / 255) + 1 \quad (13.3)$$

Os valores a serem armazenados em uma tabela são calculados por:

$$Valor_{tabela}[n] = \left\lfloor \frac{255 \times Valor_{seno}[n]}{2} \right\rfloor, \quad (13.4)$$

onde $n = 0, 1, 2, \dots, 255$. Para a composição da tabela, os valores calculados devem ser arredondados para um valor inteiro. A seguir, são apresentados os valores calculados com as eqs. 13.3 - 4 e, na fig. 13.2, o gráfico resultante.

```

Tabela_seno = [127, 130, 133, 136, 140, 143, 146, 149, 152, 155, 158, 161, 164,
    167, 170, 173, 176, 179, 182, 185, 187, 190, 193, 195, 198, 201,
    203, 206, 208, 211, 213, 215, 217, 220, 222, 224, 226, 228, 230,
    232, 233, 235, 237, 238, 240, 241, 242, 244, 245, 246, 247, 248,
    249, 250, 251, 252, 252, 253, 253, 254, 254, 254, 254, 255, 255,
    254, 254, 254, 254, 253, 252, 252, 251, 250, 250, 249, 248,
    247, 246, 244, 243, 242, 240, 239, 237, 236, 234, 232, 231, 229,
    227, 225, 223, 221, 219, 216, 214, 212, 209, 207, 204, 202, 199,
    197, 194, 191, 189, 186, 183, 180, 177, 175, 172, 169, 166, 163,
    160, 157, 154, 150, 147, 144, 141, 138, 135, 132, 129, 125, 122,
    119, 116, 113, 110, 107, 104, 100, 97, 94, 91, 88, 85, 82,
    79, 77, 74, 71, 68, 65, 63, 60, 57, 55, 52, 50, 47,
    45, 42, 40, 38, 35, 33, 31, 29, 27, 25, 23, 22, 20,
    18, 17, 15, 14, 12, 11, 10, 8, 7, 6, 5, 4, 4,
    3, 2, 2, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9,
    10, 12, 13, 14, 16, 17, 19, 21, 22, 24, 26, 28, 30,
    32, 34, 37, 39, 41, 43, 46, 48, 51, 53, 56, 59, 61,
    64, 67, 69, 72, 75, 78, 81, 84, 87, 90, 93, 6, 99,
    102, 105, 108, 111, 114, 118, 121, 124, 127];

```

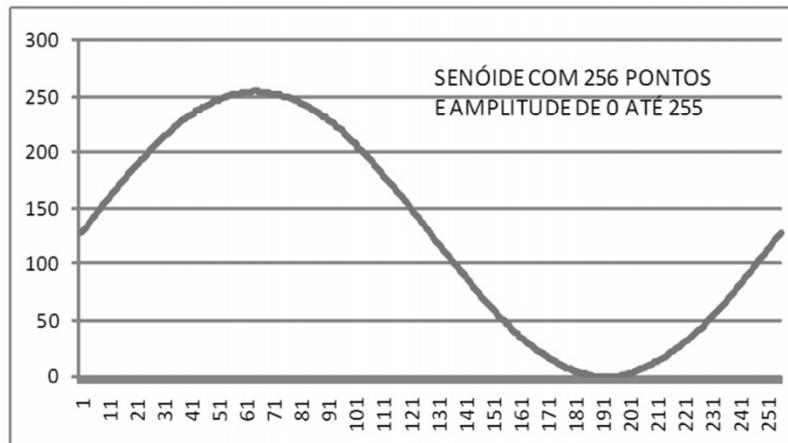


Fig. 13.2 – Senóide com 256 pontos e amplitude discreta.

A frequênciā do sinal de saída será dada pela taxa de atualização dos valores enviados para o conversor DA (Δ_t). Assim, a frequênciā do sinal de saída será dada por:

$$f = \frac{1}{255 \times \Delta_t} \quad [\text{Hz}] \quad (13.5)$$

Em resumo, um programa para criar um sinal senoidal deve ler uma tabela com os dados da amplitude do sinal e esses valores devem ser enviados a uma taxa ajustável para um conversor DA.

Exercícios:

13.1 – Elaborar um programa para gerar uma onda senoidal de acordo com o circuito da fig. 13.3. A frequência do sinal pode ser alterada por dois botões.

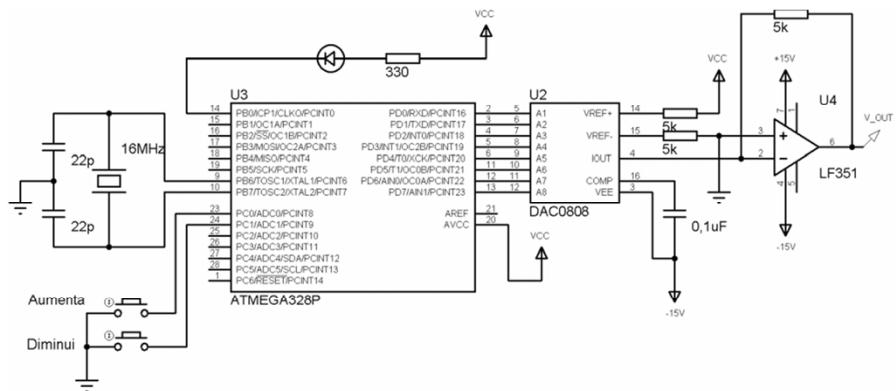


Fig. 13.3 – Circuito para gerar um sinal senoidal.

13.2 – Com base no exercício anterior, como empregar o ATmega para produzir as formas de onda triangular, dente-de-serra e quadrada?

13.2 CONVERSOR DIGITAL-ANALÓGICO COM UMA REDE R/2R

Existem inúmeros circuitos integrados dedicados para converter um número binário em seu correspondente analógico (DACs). Entretanto, muitas vezes é possível empregar um arranjo adequado de resistores para substituí-los. A rede R/2R é uma solução usual, muito empregada pela facilidade de sua confecção. Na fig. 13.4, essa rede é apresentada para uma entrada de 8 bits.

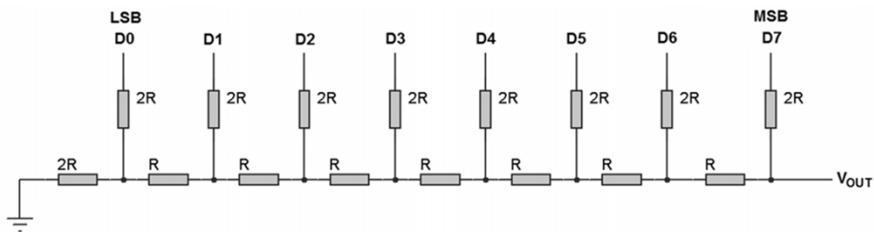


Fig. 13.4 – Rede de resistores R/2R para converter um sinal digital de 8 bits em um sinal analógico.

A tensão de saída da rede R/2R de N entradas é dada por:

$$V_{\text{OUT}} = \frac{V_{\text{REF}} \times \text{Valor}_D}{2^N}, \quad (13.6)$$

onde V_{REF} é a tensão de referência (tensão de saída dos bits ou tensão de alimentação do circuito digital - VCC) e o Valor_D é o valor decimal do número binário para a conversão.

Por exemplo: supondo um número binário de 8 bits com o valor 0b01011001 (89), proveniente de um circuito digital alimentado com 3,3 V, a tensão de saída para uma rede R/2R (fig. 13.4) será de:

$$V_{\text{OUT}} = \frac{3,3 \times 89}{2^8} = \frac{293,7}{256} \cong 1,15 \text{ V}$$

A tensão do bit menos significativo é:

$$V_{\text{OUT} \text{ LSB}} = \frac{3,3 \times 1}{256} \cong 0,013 \text{ V},$$

e a tensão de final de escala:

$$V_{\text{OUT} \text{ LSB}} = \frac{3,3 \times 255}{256} \cong 3,29 \text{ V},$$

Para maior exatidão no valor convertido, é importante o emprego de resistores de precisão na rede R/2R. Essa rede não possui capacidade para fornecer corrente a outro circuito, o que torna necessário o emprego de um amplificador com alta impedância de entrada. Uma solução simples é o emprego de algum amplificador operacional (AMPOP), muito utilizado em eletrônica analógica. Na fig. 13.5, é apresentada uma configuração utilizando um amplificador operacional com ganho unitário. O interessante no uso de um AMPOP é que se pode amplificar o sinal de entrada com o emprego de uma rede adequada de realimentação.

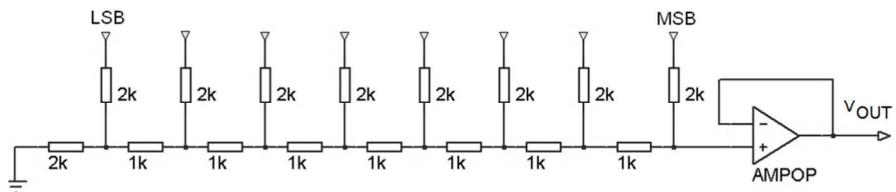
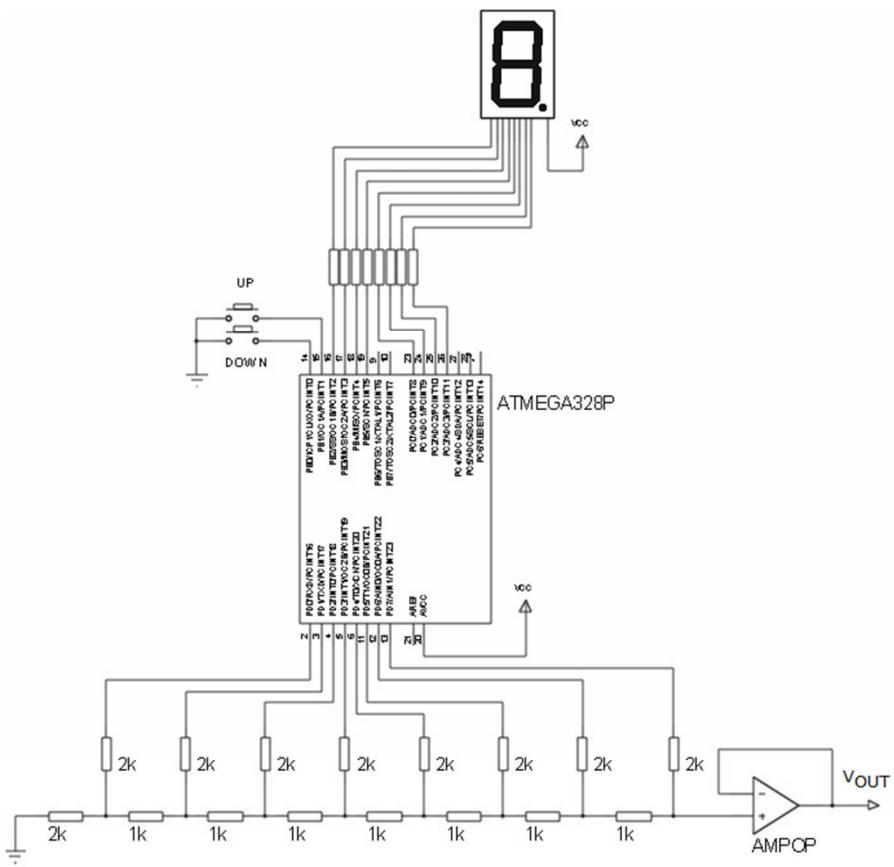


Fig. 13.5 – Circuito DA empregando uma rede R/2R com um amplificador de ganho unitário.

Exercício:

13.3 – Empregando uma rede R/2R, faça um programa para gerar uma rampa de tensão com o ATmega328. O nível da tensão de saída pode ser controlado por dois botões e um *display* de 7 segmentos pode sinalizar o nível de tensão de saída, conforme fig. 13.6.



13.3 CONVERSOR DIGITAL-ANALÓGICO COM UM SINAL PWM

O valor médio de um sinal PWM pode ser obtido com o emprego de um filtro³⁶. Isso permite criar um conversor digital-analógico simples, entretanto, de baixa precisão. O circuito da fig. 13.7 pode ser empregado para essa função.

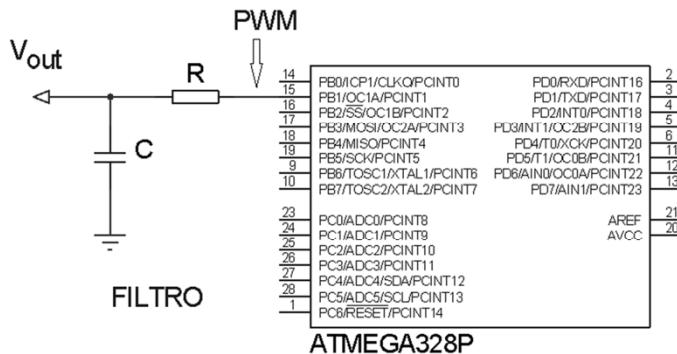


Fig. 13.7 – Conversor DA empregando um sinal PWM.

O circuito acima utiliza um filtro passa baixa de primeira ordem, cuja frequência de corte é dada por:

$$f_c = \frac{1}{2\pi RC} \quad (13.7)$$

Como o filtro de primeira ordem é um filtro ‘suave’, é importante que a frequência do sinal PWM seja pelo menos 10 vezes maior que a frequência de corte do filtro, evitando assim, uma grande atenuação do sinal. Portanto, deve ser respeitada a condição:

$$f_{PWM} > 10 \times f_c \quad (13.8)$$

³⁶ Ver o *application note* da Microchip: *D/A Conversion Using PWM and R-2R Ladders to Generate Sine and DTMF Waveform*. Para maiores detalhes sobre filtros ver: NILSSON, James W.; RIEDEL, Susan A. **Circuitos Elétricos**. 6 ed. Rio de Janeiro: LTC, 2003. e SCHERZ, Paul. **Practical Electronics for Inventors**. 2 ed. New York, McGraw Hill, 2007.

Um sinal filtrado de melhor qualidade pode ser obtido com o uso de filtros de ordem superior, como exemplificado na fig. 13.8, na qual é utilizado um transistor trabalhando na região ativa para o controle de uma carga. Para tal, emprega-se um filtro de segunda ordem: dois filtros passa baixa de primeira ordem ligados em série. Filtros de melhor qualidade exigirão um número maior de elementos passivos (resistores, capacitores e/ou indutores) e o emprego de amplificadores para evitar a atenuação do sinal (filtros ativos³⁷).

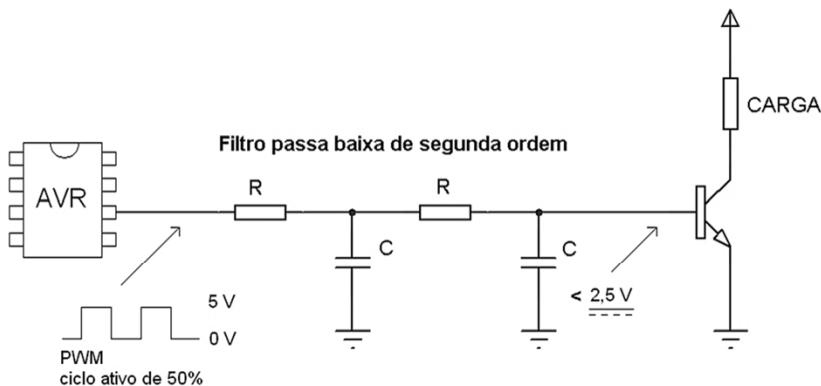


Fig. 13.8 – Transistor controlado por um sinal PWM para o acionamento de uma carga.

Exercícios:

13.4 – Elaborar um circuito que funcione como um conversor DA para um sinal PWM. Desenvolva um programa para o teste.

13.5 - Baseado no conceito de valor médio, empregue o PWM do ATmega para criar um sinal senoidal com frequência de 10 Hz. Obs.: para uma precisão adequada, a frequência do sinal PWM deve ser no mínimo 10 vezes maior que a frequência do sinal gerado.

Como alterar a frequência do sinal gerado?

³⁷ Consultar PERTENCE Jr., Antônio. **Eletrônica Analógica - Amplificadores Operacionais e Filtros Ativos**. 7 ed. Porto Alegre: Artmed, 2012.

13.4 SINAL PWM PARA UM CONVERSOR CC-CC (BUCK)

As fontes chaveadas substituem eficientemente as fontes lineares. Apresentam menor tamanho, com circuitos compactos e menor perda energética. Elas também são conhecidas como conversores CC-CC. Sua característica mais marcante é não empregar os pesados e grandes transformadores utilizados nas fontes lineares. Na figura 13.9, é exemplificado a topologia de um conversor CC-CC simples, o Buck. O objetivo é ilustrar a sua forma de funcionamento e como um microcontrolador poderia ser utilizado no seu projeto.

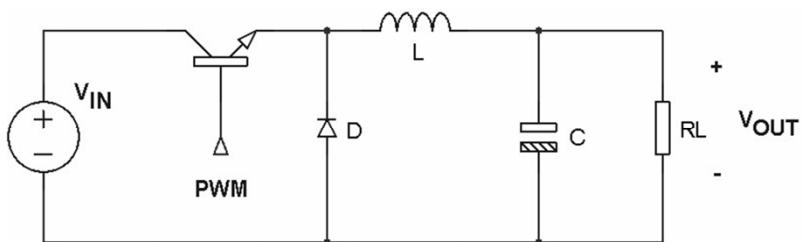


Fig. 13.9 – Conversor Buck.

O conversor CC-CC altera o nível de tensão contínua de saída a partir de uma tensão contínua de entrada. Por exemplo, na fig. 13.9 a tensão V_{IN} poderia ser resultante da retificação direta de um sinal senoidal, filtrada por um capacitor. Na primeira etapa de funcionamento do conversor Buck, o transistor está conduzindo, funcionando como chave transistorizada trabalhando na saturação. Nessa etapa, uma corrente elétrica crescente percorre o indutor carregando o capacitor e uma tensão é aplicada na resistência de carga RL . O diodo, por sua vez, encontra-se reversamente polarizado. Na segunda etapa de funcionamento, o transistor está desligado, e o indutor, dada a força contra eletromotriz, continua injetando corrente na carga, só que agora essa corrente é decrescente e o diodo está em condução. Desta forma, o diodo permite a circulação da corrente de carga enquanto o transistor estiver desligado. O resultado é uma tensão média sobre a resistência de carga. A grosso modo, pode-se dizer que o

circuito LC funciona como um filtro, onde o indutor retira a ondulação da corrente e o capacitor retira a ondulação da tensão.

Considerando-se um sinal PWM para o controle da chave transistorizada do conversor Buck, a tensão de saída é dada por:

$$V_{OUT} = \frac{T_{Ativo}}{T} \times V_{IN} \quad (13.9)$$

onde T_{Ativo} é o ciclo ativo do sinal PWM e T é o período do sinal PWM. Assim, basta ajustar o ciclo ativo do sinal PWM para controlar-se o valor da tensão de saída do conversor. A razão ciclica, a relação entre o ciclo ativo e o período é calculado por:

$$D = \frac{T_{Ativo}}{T} \quad \text{ou} \quad D = \frac{V_{OUT}}{V_{IN}} \quad (13.10)$$

O dimensionamento do indutor para o conversor segue a seguinte equação:

$$L = \frac{V_{IN}(1-D)D}{f \times 0,4I_{carga}} \quad [\text{H}] \quad (13.11)$$

onde f é a frequência do sinal PWM e I_{carga} é a corrente máxima que pode ser exigida pela carga.

Por sua vez, o capacitor pode ser dimensionado utilizando-se:

$$C = \frac{0,4I_{carga}}{2\pi \times f \times 0,01V_{OUT}} \quad [\text{F}] \quad (13.12)$$

Exemplo:

Determinar os valores de L, C e do ciclo ativo para um conversor Buck com as seguintes especificações:

- ATmega328 trabalhando a 20 MHz, TC0 no modo PWM rápido com *prescaler* = 8.
- $V_{IN} = 80 \text{ V}$, $V_{OUT} = 12 \text{ V}$, $I_{carga} = 1 \text{ A}$.

Resposta:

Frequência do sinal PWM:

$$T_{PWM} = \frac{1}{20 \text{ MHz}} \times 256 \times 8 \quad T_{PWM} = 102,4 \mu\text{s}$$

$$f_{PWM} = \frac{1}{102,4 \mu\text{s}} \cong 9,766 \text{ kHz}$$

Cálculo do ciclo ativo do sinal PWM:

$$D = \frac{V_{OUT}}{V_{IN}} = \frac{12V}{80V} \quad D = 0,15$$

$$T_{Ativo} = D \times T \quad T_{Ativo} = 0,15 \times 102,4 \mu\text{s} \quad T_{Ativo} = 15,36 \mu\text{s}$$

Considerando-se o registrador OCR0A para ajuste do ciclo ativo e saída do sinal PWM (pino OC0A do ATmega), resulta:

$$OCR0A = \frac{255 \times 15,36 \mu\text{s}}{102,4 \mu\text{s}} \cong 38$$

Cálculo do indutor (eq. 13.11):

$$L = \frac{80(1-0,15)0,15}{9766 \times 0,4 \times 1} \quad L \cong 2,6 \text{ mH}$$

Cálculo do capacitor (eq. 13.12):

$$C = \frac{0,4 \times 1}{2\pi \times 9766 \times 0,01 \times 12} \quad C \cong 54 \mu\text{F}$$

No projeto de um conversor CC-CC, costuma-se empregar chaves transistorizadas MOSFET, pois essas apresentam baixas perdas (aquecimento). O principal problema no projeto desse tipo de conversor é a confecção do indutor. A solução pode ser encontrar um valor comercial que satisfaça os requisitos do projeto.

É importante notar que muitos detalhes para o projeto do conversor Buck foram suprimidos do projeto com o microcontrolador, tais como: a adequação dos níveis de tensão do sinal PWM para os níveis de

acionamento da chave transistorizada e o controle em malha fechada para o ajuste da tensão de saída.

Exercícios:

13.6 – Refaça os cálculos do exemplo anterior com um *prescaler* igual a 1 para o TCO.

13.7 – Consulte referências bibliográficas adequadas para a revisão e melhor compreensão do equacionamento para um conversor Buck.

13.8 – Quais os tipos de conversores CC-CC utilizados para elevar a tensão?

13.9 – Como seria um circuito microcontrolado para o controle da tensão de saída de um conversor Buck? Considerando a leitura dessa tensão com o AD do microcontrolador, quais os tipos de controle que poderiam ser feitos?

13.5 SINAL PWM PARA UM CONVERSOR CC-CA

Os inversores ou conversores CC-CA são conversores estáticos que transformam energia elétrica de corrente contínua em energia elétrica de corrente alternada.

Como aplicações dos inversores, pode-se citar:

- Geração de corrente alternada de 400 Hz em aeronaves.
- Sistemas de alimentação de instrumentação.
- Fontes de alimentação ininterruptas para computadores (*nobreaks*).
- Aquecimento indutivo.
- Iluminação fluorescente em frequências elevadas ('*Ballast Eletrônico*').
- Acionamento de motores CA com frequência e tensões variáveis.
- Saídas de linhas de transmissão em corrente contínua.

Um dos tipos mais simples de inversor é a ponte monofásica, como exemplificado na fig. 13.10. A tensão na carga é ajustada pelo acionamento coordenado entre os transistores da ponte. Para a produção de uma tensão senoidal é necessário o uso de um filtro adequado na saída da ponte.

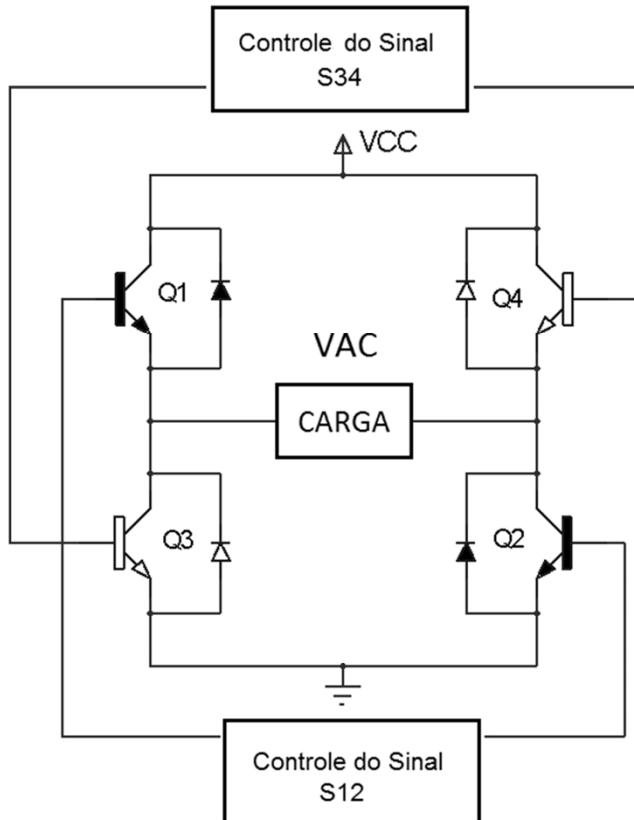


Fig. 13.10 – Conversor CC-CA monofásico.

O funcionamento da ponte é feito alternando-se o acionamento das chaves transistorizadas, primeiro Q1 e Q2 conduzem, depois Q3 e Q4. Entretanto, os transistores não são capazes de ligar e desligar instantaneamente. Se os transistores Q3 e Q4 forem acionados por um comando complementar ao de Q1 e Q2, um par será levado ao corte no

mesmo momento que o outro entra em condução, ou seja, haverá um curto período de tempo em que um dos transistores de um ramo não estará completamente bloqueado quando o outro entrar em condução, ocasionando um curto-circuito na fonte de alimentação. Assim, na prática, é incluído um tempo morto (t_M) entre os comandos complementares dos transistores, um pequeno período de tempo onde nenhum transistor da ponte estará conduzindo, evitando que um transistor entre em condução antes que o outro do mesmo ramo tenha entrando em corte.

Em controles analógicos, o acionamento dos transistores é controlado por sinais gerados pela comparação de uma onda triangular de alta frequência com uma onda senoidal de baixa frequência. A onda triangular determina a frequência de chaveamento e é chamada portadora; a onda senoidal chamada de moduladora, determina a frequência da tensão na carga.

Na fig. 13.11, é ilustrado o processo de inversão a dois níveis. O sinal de saída de um comparador tendo a onda triangular aplicada a sua entrada inversora e a onda senoidal na entrada não inversora produz o sinal de controle SC (fig. 13.11b). Após o devido condicionamento, o sinal de controle, neste caso S12 da fig. 13.11c, pode ser usado para comandar as chaves transistorizadas Q1 e Q2 do conversor da fig. 13.10. O sinal complementar de S12, por sua vez, pode ser utilizado para comandar Q3 e Q4. A tensão na carga excursiona de +VCC a -VCC e possui forma de onda similar ao sinal de controle SC.

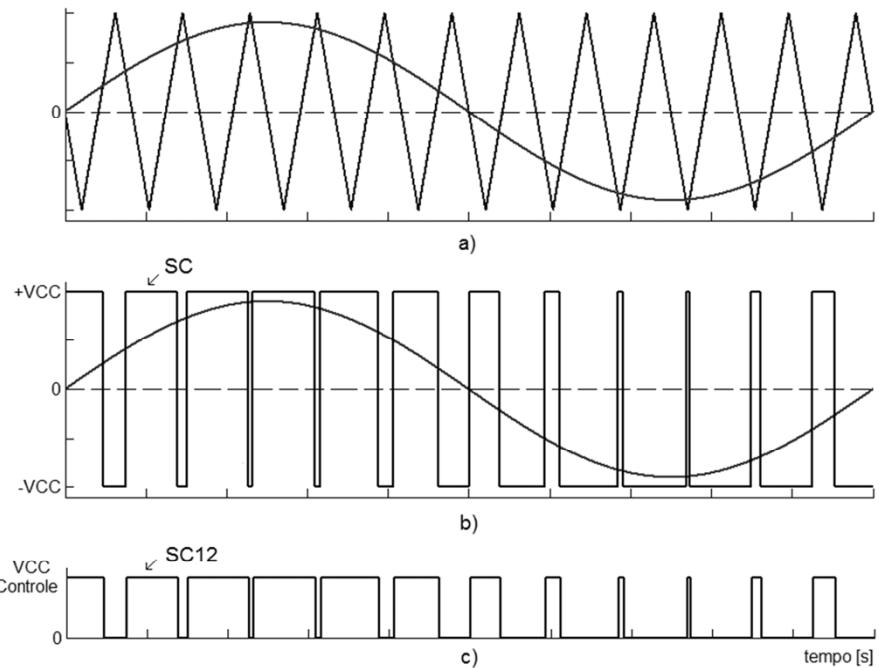


Fig. 13.11 – Modulação PWM a dois níveis: a) sinal senoidal e triangular, b) tensão PWM resultante para gerar a senóide desejada e c) sinal de controle dos transistores Q1 e Q2.

O Atmega328 pode gerar os sinais de controle nas saídas OCnA e OCnB dos módulos PWM dos seus temporizadores/contadores configurados no modo PWM de fase corrigida. Para isso, basta variar o registrador de comparação OCRnA senoidalmente e atualizá-lo por software na interrupção por estouro do temporizador/contador escolhido, gerando assim a portadora. Uma das saídas deve ser configurada no modo inversor e a outra no modo não-inversor.

Os sinais de controle para o acionamento dos transistores com o devido tempo morto são mostrados na fig. 13.12. O sinal de acionamento do par Q1/Q2 é gerado na saída OC1A configurada no modo não inversor; por sua vez, o sinal de controle para Q3 e Q4 é gerado na saída OC1B configurada no modo inversor. Observa-se que o tempo morto é gerado pela pequena diferença entre os valores de OCR1A e OCR1B, ou seja:

$$OCR1B = OCR1A + \Delta TCNT1 \quad (13.13)$$

Por inspeção na fig. 13.12, obtém-se a expressão para o cálculo do tempo morto:

$$t_M = \frac{\Delta TCNT1}{2 \cdot f_{PWM} \cdot TOP} \quad (13.14)$$

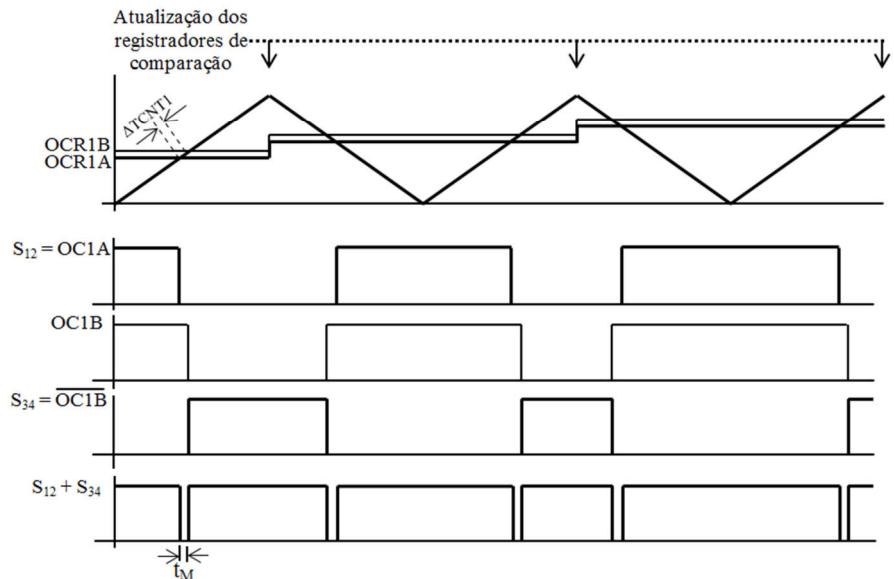


Fig. 13.12 – Processo de geração dos sinais de controle da modulação PWM a dois níveis pelo ATmega.

Obviamente, um inversor exige um projeto relativamente complexo, com o emprego de vários componentes eletrônicos trabalhando em harmonia de acordo com a topologia do conversor empregado.

Exercícios:

13.10 – Como funciona a modulação PWM senoidal mais simples que pode ser empregada em um inversor?

13.11 – Como funciona a modulação PWM senoidal a 3 níveis?

13.12 – Com funcionam e para que servem os IGBTs?

13.13 – Pesquise um circuito inversor completo para um inversor monofásico.

14. SPI

Neste capítulo, são apresentados a interface serial periférica do ATmega328 (SPI – *Serial Peripheral Interface*) e dois exemplos de dispositivos com SPI: um sensor de temperatura, o TC72, e um cartão de memória *flash*, o SD *card*.

A interface SPI é utilizada por uma infinidade de circuitos integrados, como por exemplo: conversores DAs e ADs, memórias *flash* e EEPROM, relógios de tempo real, sensores de temperatura e pressão, potenciômetros digitais, LCDs e telas sensíveis ao toque. No ATmega, além da funcionalidade usual, a SPI também é utilizada para a gravação da memória do microcontrolador (gravação *in-system*) como apresentado no capítulo 23.

A SPI é um padrão de comunicação de dados serial criado pela Motorola que opera no modo *full duplex*. Não se limita a palavras de 8 bits, assim podem ser enviadas mensagens de qualquer tamanho com conteúdo e finalidade arbitrários. A SPI é um protocolo muito simples com taxas de operação superiores a 20MHz. É um protocolo serial síncrono proposto para ser usado como um padrão para estabelecer a comunicação entre microcontroladores e periféricos. Os dispositivos no protocolo SPI são classificados como mestre ou escravos e são empregadas 4 vias para a comunicação:

- MOSI (*Master Out – Slave In*): saída de dados do mestre, entrada no escravo.
- MISO (*Master In – Slave Out*): entrada de dados no Mestre, saída do escravo.
- SCK: *clock serial*, gerado pelo mestre
- \overline{SS} (*Slave Select*) - seleção do escravo, ativo em zero.

Uma transferência de dados SPI é iniciada pelo dispositivo mestre. Ele é o responsável por gerar o sinal SCK para a transferência de dados.

Em um sistema SPI, apenas um dispositivo é configurado como mestre, os outros dispositivos são configurados como escravos. Na fig. 14.1, é apresentada uma conexão onde existe apenas um dispositivo escravo.

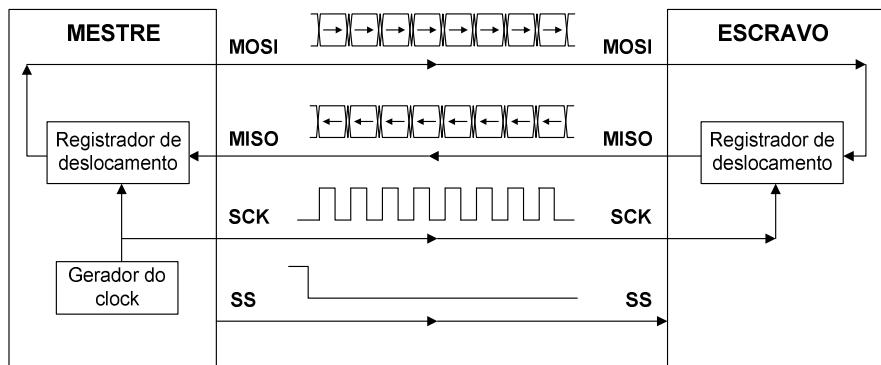


Fig. 14.1 – Conexão SPI entre um mestre e um único escravo.

Em sistemas com mais de um dispositivo escravo são possíveis dois tipos de conexão:

- Seleção individual de cada escravo: o mestre seleciona individualmente o escravo com que deseja comunicar-se, existirá um pino \overline{SS} dedicado para cada escravo. Desta forma, um dispositivo escravo não interfere no outro (fig. 14.2).
- Seleção coletiva dos escravos: os escravos são conectados em um grande anel. A saída de um escravo é conectada a entrada de outro e o circuito de dados é fechado com o mestre. O mestre seleciona todos os escravos ao mesmo tempo através de um pino \overline{SS} comum (fig. 14.3).

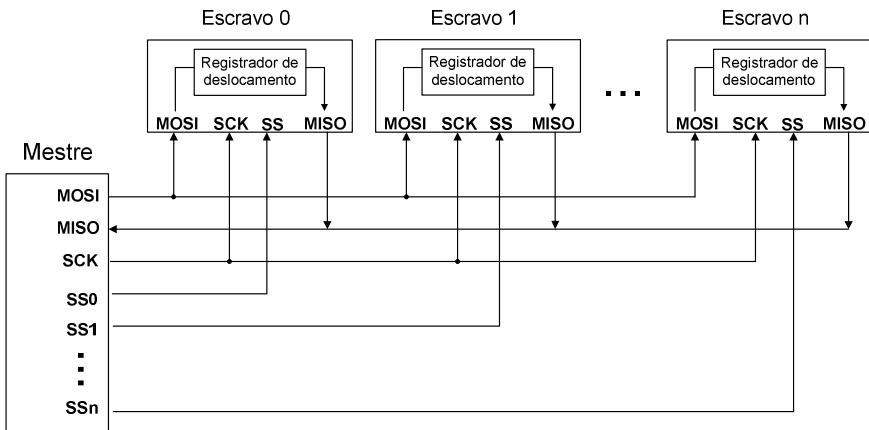


Fig. 14.2 – Conexão entre o mestre e vários escravos: seleção individual.

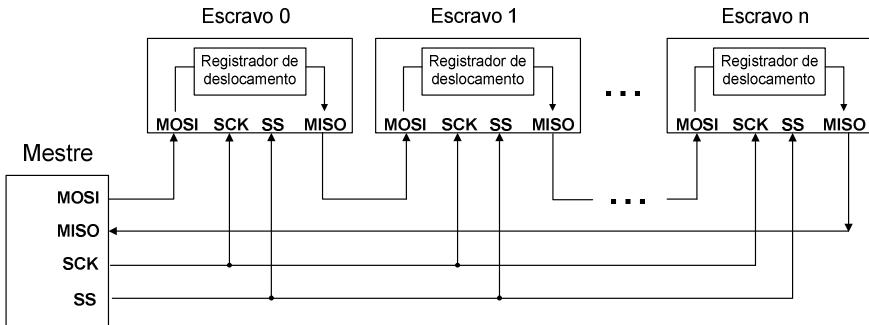


Fig. 14.3 – Conexão entre o mestre e vários escravos: seleção coletiva com ligação em anel.

As principais vantagens da SPI são:

- Protocolo simples, fácil de programar e, caso o microcontrolador não possua o módulo SPI, é fácil implementá-lo através da programação.
- Interface simples, sem pinos bidirecionais.
- Frequência de comunicação elevada, maior que 20 MHz; depende apenas das características dos dispositivos empregados.

- Dados trafegam em modo *full-duplex* (envia e recebe ao mesmo tempo).

Suas principais desvantagens são:

- A forma de transferência e amostragem de dados varia de acordo com o dispositivo.
- O número de bytes transmitidos para a realização da comunicação pode variar de acordo com os dispositivos empregados (protocolo da comunicação).
- Em sistemas com múltiplos escravos, ou a transmissão torna-se menos eficiente, caso da comunicação em anel, ou são empregados muitos pinos para a seleção dos escravos ($3 + n$, onde n representa o número de escravos).

14.1 SPI DO ATMEGA328

A Interface Serial Periférica no ATmega328 pode trabalhar com até metade da velocidade de trabalho da CPU, podendo chegar a 10 MHz, e possui as seguintes características:

- *Full-duplex* (envia e recebe dados ao mesmo tempo), transferência síncrona de dados com 3 vias.
- Operação mestre ou escravo.
- Escolha do bit a ser transferido primeiro: bit mais significativo (MSB) ou menos significativo (LSB).
- Várias taxas programáveis de *clock*.
- Sinalizador de interrupção ao final da transmissão.
- Sinalizador para proteção de colisão de escrita.
- *Wake-up* do modo *Idle*.
- Velocidade dupla no modo *Master* ($f_{osc}/2$).

O sistema é composto por dois registradores de deslocamento e um gerador mestre de *clock*. O mestre SPI inicializa o ciclo de comunicação quando coloca o pino \overline{SS} (*Slave Select*) do escravo desejado em nível baixo.

Mestre e escravo preparam o dado a ser enviado nos seus respectivos registradores de deslocamento e o mestre gera os pulsos necessários de *clock* no pino SCK para a troca de dados (ao mesmo tempo um bit é enviado e outro é recebido). Os dados são sempre deslocados do mestre para o escravo no pino MOSI (*Master Out – Slave In*) e do escravo para o mestre no pino MISO (*Master In – Slave Out*). Após cada pacote de dados, o mestre sincroniza o escravo colocando o pino \overline{SS} em nível alto.

Quando configurada como mestre, a interface SPI não tem controle automático sobre o pino \overline{SS} . Isso deve ser feito por software, antes da comunicação começar. Quando se escreve um byte no registrador de dados da SPI, a geração do *clock* inicia automaticamente e o hardware envia os oito bits para o escravo. Após esse envio, o *clock* da SPI para, ativando o aviso de final de transmissão (*flag SPIF*). Se o bit de interrupção da SPI estiver habilitado (*SPIE*) no registrador SPCR, uma interrupção será gerada. O mestre pode continuar a enviar o próximo byte, escrevendo-o no registrador SPDR, ou sinalizar o final da transmissão (\overline{SS} em nível alto). O último byte de chegada é mantido no registrador de armazenagem (*buffer*).

O sistema possui um registrador de transmissão e um duplo de recepção. Isso significa que o byte a ser transmitido não pode ser escrito no registrador de dados da SPI antes do final da transmissão do byte. Quando um dado é recebido, ele deve ser lido do registrador SPI antes do próximo dado ser completamente recebido, caso contrário, o primeiro dado é perdido.

No modo escravo, para garantir a correta amostragem do sinal de *clock*, os períodos mínimo e máximo do *clock* recebido devem ser maiores que 2 ciclos de *clock* da CPU.

Quando o modo SPI é habilitado, a direção dos pinos MOSI, MISO, SCK e \overline{SS} é ajustada conforme tab. 14.1.

Tab. 14.1 – Direção dos pinos para a SPI quando habilitada.

Pino	Direção, SPI Mestre	Direção, SPI Escravo
MOSI	definido pelo usuário	entrada
MISO	entrada	definido pelo usuário
SCK	definido pelo usuário	entrada
SS	definido pelo usuário	entrada

O código a seguir mostra funções para inicializar a SPI como mestre e executar uma transmissão simples.

```
//===================================================================== //
//Funções para inicializar a SPI no modo Mestre e transmitir um dado   //
//===================================================================== //
void SPI_Mestre_Inic()
{
    DDRB = (1<<PB5)|(1<<PB3); //ajusta MOSI e SCK como saída, demais
                                //pinos como entrada
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0); //habilita SPI, Mestre,
                                                //taxa de clock clk1/16
}
//-----
void SPI_Mestre_Transmit(char dado)
{
    SPDR = dado; //inicia a transmissão
    while(!(SPSR & (1<<SPIF))); //espera a transmissão ser completada
}
//=====================================================================
```

O código a seguir mostra funções para inicializar a SPI como escravo e executar uma recepção simples.

```
//===================================================================== //
//      Funções para inicializar a SPI no modo Escravo e receber um dado  //
//===================================================================== //
void SPI_Escravo_Inic( )
{
    DDRB = (1<<PB4); //ajusta o pino MISO como saída, demais pinos como entrada
    SPCR = (1<<SPE); //habilita SPI
}
//-----
char SPI_Escravo_Recebe( )
{
    while(!(SPSR & (1<<SPIF))); //espera a recepção estar completa
    return SPDR; //retorna o registrador de dados
}
//=====================================================================
```

SPCR – SPI Control Register

Bit	7	6	5	4	3	2	1	0
SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
Lê/Escr.	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 7 – SPIE – SPI Interrupt Enable

Quando este bit estiver em 1, a interrupção da SPI é executada se o bit SPIF do registrador SPSR também estiver em 1. O bit I do registrador SREG deve estar habilitado.

Bit 6 – SPE – SPI Enable

Bit para habilitar a SPI.

Bit 5 – DORD – Data Order

Em 1, transmite-se primeiro o LSB, em 0 transmite o MSB.

Bit 4 – MSTR – Master/Slave Select

Em 1, seleciona o modo mestre; caso contrário, o modo escravo. Se o pino \overline{SS} é configurado como entrada e estiver em nível zero enquanto MSTR estiver ativo, MSTR será limpo e o bit SPIF do SPSR será ativo. Então, será necessário ativar MSTR para reabilitar o modo mestre.

Bit 3 – CPOL – Clock Polarity

Este bit em 1 mantém o pino SCK (quando inativo) em nível alto; em zero, mantém o pino em nível baixo.

Bit 2 – CPHA – Clock Phase

Este bit determina se o dado será coletado na subida ou descida do sinal de *clock*.

Bits 1:0 – SPR1:0 – SPI Clock Rate Select 1 e 0

Estes dois bits controlam a taxa do sinal SCK quando o microcontrolador é configurado como mestre; sem efeito quando configurado como escravo (tab. 14.2).

Tab. 14.2 – Seleção da frequência de operação para o modo mestre (f_{osc} = freq. de operação da CPU).

SPI2X	SPR1	SPR0	Frequência do SCK
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

SPSR – SPI Status Register

Bit	7	6	5	4	3	2	1	0
SPSR	SPIF	WCOL	-	-	-	-	-	SPI2X
Lê/Escrive	L	L	L	L	L	L	L	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 7 – SPIF – SPI Interrupt Flag

Colocado em 1 quando uma transferência serial for completada.

Bit 6 – WCOL – Write COLision Flag

É ativo se o registrador SPDR é escrito durante uma transmissão.

Bit 0 – SPI2X – Double SPI Speed Bit

Em 1 duplica a velocidade de transmissão quando no modo mestre. No modo escravo, a SPI trabalha garantidamente com $f_{osc}/4$.

SPDR – SPI Data Register

Bit	7	6	5	4	3	2	1	0
SPDR	MSB							LSB
Lê/Escrive	L/E							
Valor Inicial indefinido	X	X	X	X	X	X	X	X

Escrever neste registrador inicia uma transmissão. Ao lê-lo, carrega-se o conteúdo do *buffer* do registrador de entrada.

Modos de operação

Existem 4 combinações de fase e polaridade de *clock* com respeito aos dados seriais, determinadas pelos bits de controle CPOL e CPHA, conforme tab. 14.3 e figs. 14.4-5.

Tab. 14.3 – Funcionalidade dos bits CPOL e CPHA.

Configuração	Borda do SCK	Borda do SCK	MODO SPI
CPOL = 0, CPHA = 0	amostragem (subida)	ajuste (descida)	0
CPOL = 0, CPHA = 1	ajuste (subida)	amostragem (descida)	1
CPOL = 1, CPHA = 0	amostragem (descida)	ajuste (subida)	2
CPOL = 1, CPHA = 1	ajuste (descida)	amostragem (subida)	3

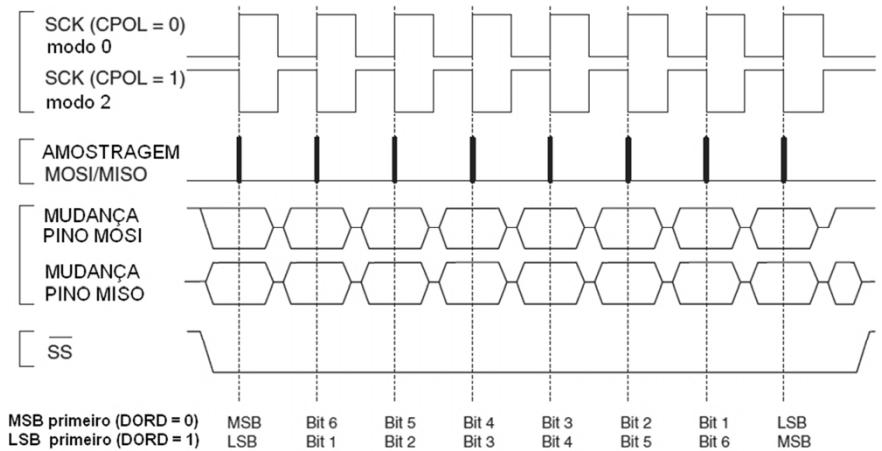


Fig. 14.4 – Formato da transferência SPI com CPHA=0.

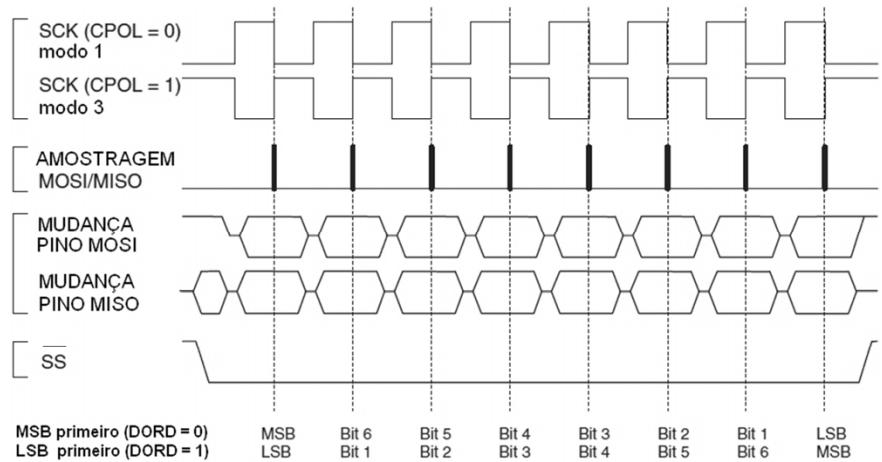


Fig. 14.5 – Formato da transferência SPI com CPHA=1.

14.2 SENSOR DE TEMPERATURA TC72

O circuito integrado TC72 é um termômetro digital com interface SPI. É capaz de medir temperaturas entre -55 °C e +125 °C. Suas principais características são:

- 10 bits de resolução (0,25 °C/bit);
- precisão de ± 3 °C de -55 °C a +125 °C ou ± 2 °C de -40 °C a +85 °C;
- tensão de alimentação entre 2,65 V e 5,5 V;
- tempo de conversão de 150 ms;
- modo de conversão contínuo ou única conversão;
- frequência máxima de trabalho de 7,5 MHz;
- consumo de 150 μ A no modo de conversão contínua;
- encapsulamento de 8 pinos MSOP ou DFN (3 × 3 mm).

A temperatura é representada por um conjunto de 10 bits em complemento de dois com uma resolução de 0,25 °C por bit menos significativo. É escalonada de -128 °C a +127 °C e armazenada em dois registradores de 8 bits, cujo bit mais significativo indica se a temperatura é negativa ou positiva. Na tab. 14.4, são apresentados exemplos para valores convertidos e, nas tabs. 14.6-7, os registradores para o trabalho e o significado dos seus bits.

Tab. 14.4 – Formato para alguns valores de temperatura.

Temperatura	Valor binário MSByte [*] /LSByte [*]
+125 °C	0111 1101 / 0000 0000
+25 °C	0001 1001 / 0000 0000
+0,5 °C	0000 0000 / 1000 0000
+0,25 °C	0000 0000 / 0100 0000
0 °C	0000 0000 / 0000 0000
-0,25 °C	1111 1111 / 1100 0000
-25 °C	1110 0111 / 0000 0000
-55 °C	1100 1001 / 0000 0000

^{*} MSByte = Byte Mais Significativo.

^{*} LSByte = Byte Menos Significativo.

Tab. 14.5 – Registradores do TC72 (nos registradores de temperatura são apresentados os pesos dos bits).

Registrador	End. Leitura	End. Escrita	D7	D6	D5	D4	D3	D2	D1	D0	Valor na Inicializ.
Controle	0x00	0x80	0	0	0	OS*	0	0	0	SHDN**	0x05
Temperatura LSByte	0x01	-	2^{-1}	2^{-2}	0	0	0	0	0	0	0x00
Temperatura MSByte	0x02	-	Sinal	2^6	2^5	2^4	2^3	2^2	2^1	2^0	0x00

*OS = One-Shot (bit de conversão única).

**SHDN = Shutdown (bit para desligar o TC72, deixá-lo em Standby, consumo de 1µA).

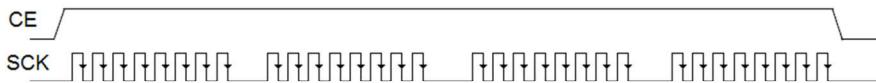
Tab. 14.6 – Modos de operação de acordo com os bits OS e SHDN.

Modo de Operação	OS	SHDN
Conversão contínua	0	0
Desliga	0	1
Conversão contínua	1	0
Conversão única	1	1

O trabalho com o TC72 deve ser feito da seguinte forma:

- A entrada CE (habilitação) do TC72 deve ser colocada em 1 lógico para habilitar a transferência SPI (obs.: nível lógico contrário ao usual para habilitação pela SPI).
- O dado pode ser deslocado na borda de subida ou descida do *clock*; o padrão é na borda de descida.
- A transferência de dados consiste de um byte de endereço seguido por um ou múltiplos dados (2 a 4 bytes).
- O bit mais significativo do byte de endereço determinará uma operação de leitura (bit 7 = 0) ou uma escrita (bit 7 = 1).
- Uma operação de leitura de múltiplos bytes iniciará a partir do endereço mais alto em direção ao mais baixo.
- O usuário pode enviar o endereço do byte alto para ler os dois bytes de temperatura e o byte do registrador de controle. Na fig. 14.6, é ilustrada a leitura e escrita de múltiplos bytes.

Transferência de múltiplos bytes (amostragem do dado na borda de decida do clock)



Operação de Escrita

SDI Endereço = 0x80 Byte de controle



SDO Alta Impedância

Operação de Leitura

SDI Endereço = 0x02



Fig. 14.6 – Transferência de múltiplos bytes para o TC72.

Em resumo, os passos para o trabalho com o TC72 são:

1. CE = 1 para habilitá-lo.
2. Transmitir 0x80 numa operação de escrita, mais o byte de configuração de controle (para conversão contínua = 0x00 ou 0x10). Após, CE = 0 para finalizar a configuração.
3. Novamente CE = 1.
4. Transmitir 0x02.
5. Transmitir qualquer valor para ler o byte mais significativo da temperatura.
6. Transmitir qualquer valor para ler o byte menos significativo da temperatura. Se desejado pode ser enviado mais um byte para ler o byte de controle.
7. CE = 0 para desabilitar o TC72, permitindo uma nova conversão.
8. Esperar pelo menos 150 ms e voltar ao passo 3.

A seguir, é apresentado um programa para o trabalho com o TC72. A temperatura com uma resolução de 1°C é mostrada em um LCD 16 × 2 (o resultado pode ser visto na fig. 14.7).

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL //define a frequência do microcontrolador - 16MHz

#include <avr/io.h> //definições do componente especificado
#include <util/delay.h> //biblioteca para o uso das rotinas de _delay_ms e _delay_us()
#include <avr/pgmspace.h> //para o uso do PROGMEM, gravação de dados na memória flash

//Definições de macros para o trabalho com bits

#define set_bit(y,bit) (y|=(1<<bit))//coloca em 1 o bit x da variável Y
#define clr_bit(y,bit) (y&=~(1<<bit))//coloca em 0 o bit x da variável Y
#define cpl_bit(y,bit) (y^=(1<<bit))//troca o estado lógico do bit x da variável Y
#define tst_bit(y,bit) (y&(1<<bit)) //retorna 0 ou 1 conforme leitura do bit

#endif
```

TC72.c (programa principal)

```
===== //
//      Sensor Digital de temperatura TC72 com interface SPI          //
//      Resolução de amostragem para o LCD de 1 grau Centígrado       //
===== //
#include "def_principais.h"
#include "LCD.h"

#define habilita_TC72()    set_bit(PORTB,PB2)  //nível lógico do pino SS é contrário ao usual
#define desabilita_TC72()  clr_bit(PORTB,PB2)

void inic_SPI();
unsigned char SPI(unsigned char dado);
//-----
int main()
{
    unsigned char temp, tempH,tempL;
    char TEMP[3];

    DDRD = 0b11111100;//LCD
    PORTD = 0b00000011;

    inic_SPI();
    inic_LCD_4bits();

    cmd_LCD(0x80,0);
    escreve_LCD("TEMPERATURA");
    cmd_LCD(0xCE,0);
    cmd_LCD(0xDF,1);
    cmd_LCD('C',1);
```

```

habilita_TC72();      //CE=1
SPI(0x00);           //Prepara a escrita
SPI(0x00);           //Envia byte de controle, conversão continua
desabilita_TC72();   //CE=0

while (1)
{
    _delay_ms(150);

    habilita_TC72();
    SPI(0x02);          //Prepara para a leitura
    tempH=SPI(0x00);    //Lê MSB
    templ=SPI(0x00);    //Lê LSB
    desabilita_TC72();

    cmd_LCD(0xCA,0);

    if(tst_bit(tempH,7))
    {
        cmd_LCD('-',1);
        tempH = 128 - tempH;//converte para temperatura positiva
    }
    else
        cmd_LCD('+',1);

    //MSB contém o valor da temperatura com resolução de 1 grau centígrado
    temp = tempH & 0b01111111;//zera o bit de sinal
    ident_num(temp, TEMP);
    cmd_LCD(TEMP[2],1);
    cmd_LCD(TEMP[1],1);
    cmd_LCD(TEMP[0],1);
}
//-----
//Inicialização da SPI
//-----
void inic_SPI()
{
    //configuração dos pinos de entrada e saída da SPI
    DDRB = (DDRB & 0b11000011)|(0b000101100);
    SPCR = (1<<SPE)|(1<<MSTR)|(SPR1)|(1<<CPHA);
    /*habilita SPI master, modo 0, CLK = f_osc/64 (250 kHz), dado ajustado na subida e amostragem
    na descida do sinal de clock*/
}
//-----
//Envia e recebe um byte pela SPI
//-----
unsigned char SPI(unsigned char dado)
{
    SPDR = dado;           //envia um byte
    while(!(SPSR & (1<<SPIF))); //espera envio
    return SPDR;           //retorna o byte recebido
}
//-----

```

LCD.h e **LCD.c** (como apresentados no capítulo 5).

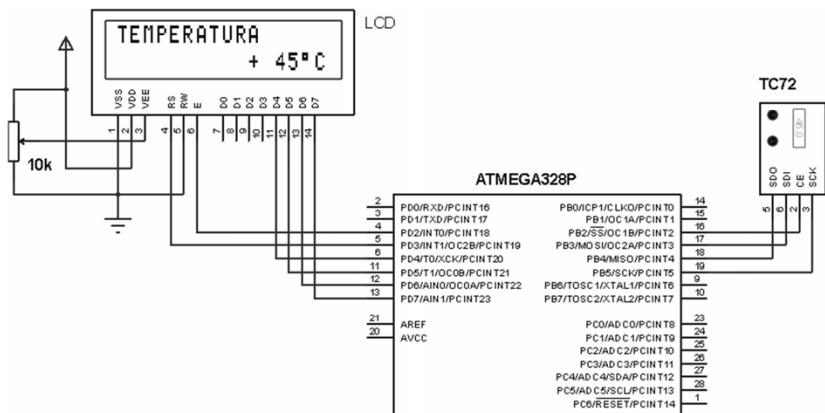


Fig. 14.7 – ATmega328 e o TC72.

14.3 CARTÃO DE MEMÓRIA SD/MMC

Os cartões de memória *flash*, como os encontrados em celulares, câmeras digitais, *tablets* e computadores, são atualmente muito utilizados. Entre a infinidade de dispositivos de memória disponíveis, os cartões mais comuns são o SD (*Secure Digital memory card*) e o MMC (*Multi Media Card*). Eles podem usar o protocolo SPI e, portanto, têm sido muito empregados em sistemas embarcados. Sua principal aplicação é no armazenamento de dados em sistemas que necessitam gravar variáveis temporais, como por exemplo, *data loggers*.

Os SD e os MMC são compatíveis entre si. Todavia, o MMC é mais fino e o seu suporte não permite o encaixe do SD; já o contrário é possível, o suporte para o SD permite o encaixe do MMC. Os SDs apresentam uma chave de segurança contra escritas acidentais, são mais populares e, ainda, são disponibilizados em tamanhos menores, como o mini SD e o micro SD (utilizado nos celulares e *tablets*). Os SDs padrão possuem capacidade de memória de até 4 GB. Todavia existem modelos que chegam até 32 GB (*High-Capacity SD Card – SDHC*). Na fig. 14.8, é apresentado a pinagem para o SD e o MCC, incluindo o diagrama esquemático para o SD.

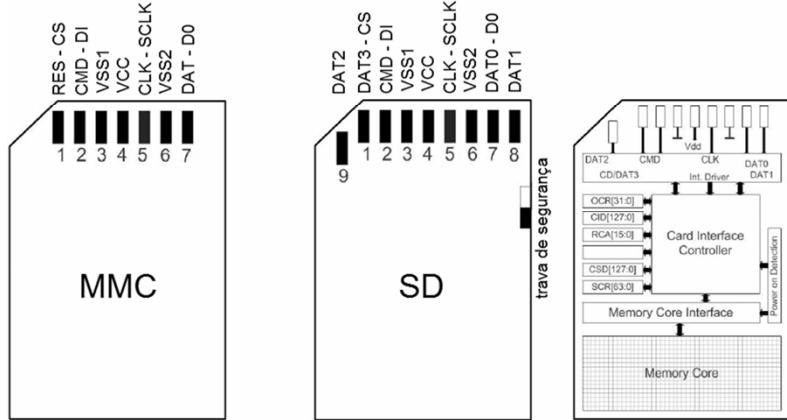


Fig. 14.8 – Pinagem para os cartões de memória MMC e SD, incluindo o diagrama esquemático para o SD (adaptado de: *SD Card Reader Using the M9S08JM60 Series Designer Reference Manual*).

A tensão de alimentação para os SD/MMCs deve estar compreendida entre 2,7 V e 3,6 V; geralmente, utiliza-se 3,3 V. Dessa forma, para sistemas alimentados com 5 V, é necessário o emprego de resistores configurados como divisores de tensão para a limitação dos sinais de tensão do microcontrolador para o cartão de memória (fig. 14.9). A corrente durante uma operação de escrita pode chegar a 100 mA.

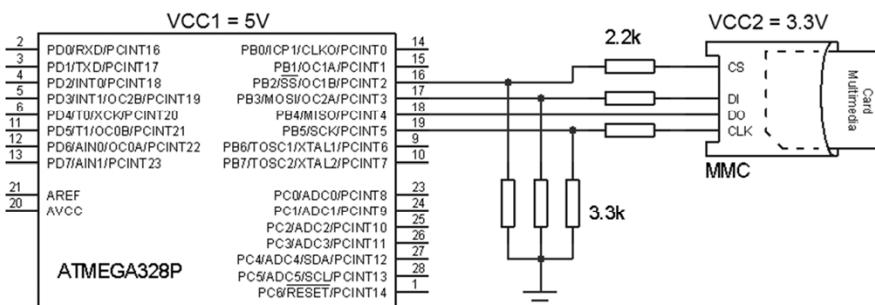


Fig. 14.9 – Circuito para adaptar os níveis lógicos de tensão de um microcontrolador alimentado com 5 V para a comunicação com um cartão SD/MMC.

Os cartões SD possuem dois modos de operação: o modo nativo - padrão, chamado modo de barramento SD, e o modo de barramento SPI (igual para o MMC). O modo nativo é bem mais complexo que o modo SPI e é empregado em sistemas que necessitam obter o máximo desempenho de escrita e leitura e podem dispor de uma programação mais complexa. Nesse modo, todos os pinos do cartão são utilizados e os dados são transmitidos com 4 pinos (D0-D3), um pino de *clock* (CLK) e um de comando (CMD). O modo SPI é o mais fácil de programar e é o preferido para uso com microcontroladores, pois utiliza apenas os pinos da SPI (MISO – DO, MOSI – DI, CLK e SS – CS), conforme apresentado na fig. 14.9.

No modo SPI, a transmissão de dados é feita por bytes de forma serial. Os comandos utilizam 6 bytes de formato, conforme fig. 14.10. Após um determinado tempo, o cartão de memória retorna um ou mais bytes de resposta, chamados R1, R2 ou R3. O índice indica qual comando será aplicado e pode ter um valor entre 0 e 63. O argumento utilizado vai depender de qual comando está sendo empregado; no caso de leitura ou escrita, conterá um endereço.

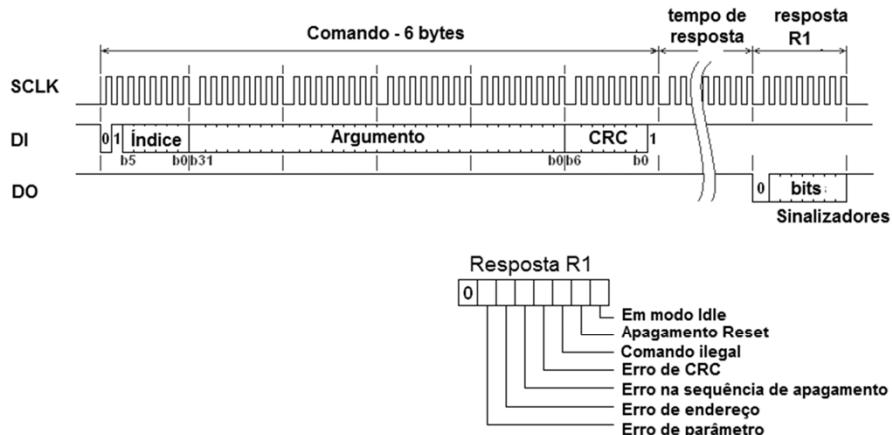


Fig. 14.10 - Formato do comando e resposta para o controle de um SD/MMC no modo SPI (adaptado de: http://elm-chan.org/docs/mmc/mmc_e.html).

Como a geração de *clock* é feita pelo dispositivo de controle (microcontrolador), esse deve ficar continuamente lendo os dados da memória até obter um valor válido. O pino DI (MOSI) deve ser mantido em nível lógico alto. Portanto, deve ser enviado o valor 0xFF na leitura de dados. Para realizar algum comando, o cartão de memória deve ser habilitado, colocando-se o pino CS em nível lógico zero. O CRC (*checksum - conferência de dados*) é opcional e, geralmente, desabilitado no modo SPI.

Existem inúmeros comandos que podem ser utilizados. Para as operações básicas são necessários apenas os seguintes:

- CMD0 (índice = 0) <GO_IDLE_STATE>, inicialização da memória, entrada no modo *Idle*. O argumento para a função é 0x00000000 e deve ser utilizado o valor 0x95 para o CRC (única vez, depois ele estará desabilitado no modo SPI, a menos que programado o contrário). Assim, o CMD0 será 0x400000000095.
- CMD1 (índice = 1) <SEND_OP_COND>, quando a memória está no modo *Idle*, ela pede a condição de operação para saber se a inicialização foi efetuada corretamente. O formato do CMD1 será 0x410000000FF (CRC não empregado = 0xFF).
- CMD16 (índice = 16) <SET_BLOCKLEN>, ajusta o comprimento do bloco de dados em bytes (512, 1024, 2048, 4096) para a escrita e leitura. O padrão é 512 e não precisa ser ajustado, a menos que o contrário seja desejado. O cartão de memória só permite a escrita e leitura em blocos.
- CMD17 (índice = 17) <READ_SINGLE_BLOCK>, leitura de um único bloco de dados. O formato de dados será 0x58xxxxxxxxx FF, onde xx é o endereço de início da leitura, a qual sempre é feita em blocos com o tamanho determinado pelo CMD16.
- CMD24 (índice = 24) <WRITE_BLOCK>, escrita de um bloco de dados, cujo tamanho é determinado pelo CMD16. O formato será 0x51xxxxxxxxx FF, onde xx é o endereço de início para a escrita do bloco de dados.

Na transmissão ou recepção do bloco de dados, sempre é transmitido primeiro o bit mais significativo (MSB), sendo que o bloco deve possuir 515 bytes (*Data Tokens*), dispostos da seguinte forma:

<1 byte indicando o inicio>:<512 bytes de dados>:<2 bytes de CRC>

O primeiro byte deve ser 0xFE, indicando o início do bloco de dados, depois seguem os 512 bytes dos dados e, por último, dois bytes do CRC dos dados. Quando este não for empregado, escreve-se 0xFFFF.

Quando o cartão de memória é ligado, ele encontra-se no modo nativo e precisa ser configurado para o modo SPI (nesse modo, o CRC é desabilitado por padrão). Os seguintes passos são necessários para inicializar corretamente um cartão SD:

- Enviar no mínimo 74 pulsos de *clock* para o cartão com o pino CS e DO em 1.
- Habilitar o cartão, fazendo CS = 0.
- Enviar o comando CMD0.
- Verificar a resposta R1 para se certificar que não existe algum bit de erro.
- Enviar repetidamente o comando CMD1, até que o bit de ‘em modo Idle’ da resposta R1 ser colocado em zero e não haver bits de erro.

Após a inicialização, a memória está pronta para o uso. Durante a inicialização, a frequência da SPI deve estar entre 100 kHz e 400 kHz. Após, pode ser aumentada para até 25 MHz para o cartão SD e 20 MHz para o MMC.

A forma básica de escrita é simples. Todavia, pode ser empregado o sistema de arquivo FAT16 para uso nos cartões SD padrão e FAT32 para cartões com capacidades maiores de memória. O emprego de um sistema de arquivos FAT permite a leitura dos dados diretamente em um computador. Entretanto, agrega maior complexidade ao software de controle.

A seguir, é apresentado um programa para escrita e leitura de um cartão SD/MMC pelo ATmega328. O programa principal escreve em três setores da memória (blocos de 512 bytes) e verifica se o último conjunto de dados escrito corresponde ao lido. Foi empregado um LED sinalizador para indicar a ocorrência de erro ou sucesso na operação. O programa analisa a ocorrência de erros na comunicação e usa variáveis para a contagem, permitindo um tempo máximo de resposta para o cartão.

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef DEF_PRINCIPAIS_H_
#define DEF_PRINCIPAIS_H_

#define F_CPU 16000000UL

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

//Definições de macros para o trabalho com bits

#define    set_bit(y,bit) (y|=(1<<bit)) //coloca em 1 o bit x da variável Y
#define    clr_bit(y,bit) (y&=~(1<<bit)) //coloca em 0 o bit x da variável Y
#define    cpl_bit(y,bit) (y^=(1<<bit)) //troca o estado lógico do bit x da variável Y
#define    tst_bit(y,bit) (y&(1<<bit)) //retorna 0 ou 1 conforme leitura do bit

#endif
```

SD_MMC_card.c (programa principal)

```
----- //
//  Programa para a escrita e leitura de um SD/MCC                         //
//  Escrita e leitura em blocos de 512 bytes                                //
----- //

#include "def_principais.h"
#include "SD_MMC.h"

#define liga_led()      set_bit(PORTD,5)      //ativo em 1
#define desliga_led()   clr_bit(PORTD,5)

/* aloca 512 bytes da RAM para poder escrever em blocos, como necessário para o
   trabalho com o SD card*/
unsigned char dados[512];

int main(void)
{
    unsigned int i;
```

```

DDRD = 1<<5; //PD5 possui o LED sinalizador de erro
desliga_led(); /*Só liga o LED se houver erro, pisca led ao final se o
funcionamento foi correto*/
inic_SPI();

//inicializa MMC - após inicialização a velocidade da SPI pode ser aumentada
if(inic_MMC()!=0)
    liga_led();

//teste para a escrita em 3 setores da memória
//-----
//setor 1
for(i=0;i<512;i++)  dados[i]=0xAA;

if(escreve_MMC(dados,0,0)!=0) //escreve 0xAA nos 512 bytes do setor 1
    liga_led();
//-----
//setor 2
for(i=0;i<512;i++)  dados[i]=0xBB;

if(escreve_MMC(dados,0,512)!=0) //escreve 0xBB nos 512 bytes do setor 2
    liga_led();
//-----
//setor 1000 = 0x7CE00
for(i=0;i<512;i++)  dados[i]=(uchar)i;

if(escreve_MMC(dados,0x0007,0xCE00)!=0) //escreve 0-511 nos 512 bytes do setor 1000
    liga_led();
//-----
//comparação da escrita apenas para o setor 1000
if(le_MMC(dados,0x0007,0xCE00)!=0) //lê as primeiras 512 posições
    liga_led();

//-----
//compara os dados escritos com os lidos
for(i=0;i<512;i++)
{
    if(dados[i]!=(uchar)i)
        liga_led(); //liga o led se o dado escrito for diferente do lido
}

//se o led não foi ligado fica piscando indicando sucesso na operação de escrita e leitura
if(!tst_bit(PORTD,5))
{
    while(1)
    {
        liga_led();
        _delay_ms(300);
        desliga_led();
        _delay_ms(300);
    }
}
//-----
while(1);
}

```

SD_MMC.h (arquivo de cabeçalho do SD_MMC.c)

```
#ifndef SD_MMC_H_
#define SD_MMC_H_

#include "def_principais.h"

typedef unsigned char uchar;
typedef unsigned int uint;

#define DI      PB3      //entrada de dados (dados da memória)
#define DO      PB4      //saída de dados (dados para a memória)
#define CLK     PB5      //clock
#define CS      PB2      //seleção da memória (habilitação)

//comandos básicos para o SD/MMC

#define habilita_MMC()    clr_bit(PORTB,CS)
#define desabilita_MMC()  set_bit(PORTB,CS)

void inic_SPI();
unsigned char SPI(unsigned char c);

void comando_MMC(uchar index, uint ArgH, uint ArgL, uchar CRC);
unsigned char resposta_MMC(uchar resposta);
unsigned char inic_MMC();
unsigned char escreve_MMC(uchar *buffer, uint enderecoH, uint enderecoL);
unsigned char le_MMC(uchar *buffer, uint enderecoH, uint enderecoL);

#endif
```

SD_MMC.c (arquivo com as funções para o trabalho com os cartões SD/MMC)

```
#include "SD_MMC.h"
//-----
//Inicialização da SPI
//-----
void inic_SPI()
{
    //configuração dos pinos de entrada e saída da SPI
    DDRB = (DDRB & 0b11000011)|(0b00101100);

    habilita_MMC(); //CS = 0

    SPCR = (1<<SPE)|(1<<MSTR)|(SPR1); /*habilita SPI master, modo 0, CLK = f_osc/64
                                            (250 kHz), 100 kHz <= f_osc <= 400 kHz*/
}

//-----
//Envia e recebe um byte pela SPI
//-----
unsigned char SPI(uchar dado)
{
    SPDR = dado;           //envia um byte
    while(!(SPSR & (1<<SPIF))); //espera envio

    return SPDR;           //retorna o byte recebido
}
```

```

//-----
//Envia um comando para o MMC - index + argumento + CRC (total de 6 bytes)
//-----
void comando_MMC(uchar index, uint ArgH, uint ArgL, uchar CRC)
{
    //INDEX - 6 bits
    SPI(index);//sempre bit 7 = 0 e bit 6 = 1, b5:b0 (index)

    //ARGUMENTO - 32 bits
    SPI((uchar)(ArgH >> 8));
    SPI((uchar)ArgH);
    SPI((uchar)(ArgL >> 8));
    SPI((uchar)ArgL);

    //CRC
    SPI(CRC); //bit 0 sempre 1
}
//-----
//Lê continuamente o MMC até obter a resposta desejada ou estourar o tempo permitido
//-----
unsigned char resposta_MMC(uchar resposta)
{
    unsigned int cont = 0xFF; //contador para espera de um máximo tempo da resposta

    do
    {
        if(SPI(0xFF)==resposta)
            return 0; //resposta normal laço acabou antes do estouro de contagem
        cont--;
    }while(cont!=0);
    return 1; //falha na resposta
}
//-----
//Inicializa o MMC para trabalhar no modo SPI
//-----
unsigned char inic_MMC()
{
    unsigned char i;

    desabilita_MMC();
    for(i=0; i<10; i++) SPI(0xFF); //80 pulsos de clock
    habilita_MMC();

    //CMD0 (GO TO IDLE STATE - RESET) - único vez que envia o checksum (0x95)
    comando_MMC(0x40,0,0,0x95);

    if (resposta_MMC(0x01)) { desabilita_MMC(); return 1;}/*retorna 1 para indicar
                                                       que houve erro (não entrou no modo Idle)*/
    desabilita_MMC();
    SPI(0xFF); //alguns clocks após o modo Idle
    habilita_MMC();

    i=0xFF;
    /*envia o CMD1 até resposta ser zero (não estar mais no modo Idle), não haver
       erro, e estar dentro do nr. máximo de repetições*/
    do
    {
        i--;
        //CMD1 (SEND_OP_COND) - verifica quando saiu do modo Idle
        comando_MMC(0x41,0,0,0xFF);
    } while ((resposta_MMC(0x00)!=0) && (i!=0));
}

```

```

//retorna 2 para indicar que houve erro não saiu do modo Idle)
if(i==0){ desabilita_MMC(); return 2;}

desabilita_MMC();
SPI(0xFF);      //alguns clocks após modo Idle

return 0;        //retorna 0 para indicar sucesso na operação
}
//-----
//Escreve a partir do endereço especificado (setor), sempre em blocos de 512 bytes
//Os endereços podem ser 0, 512, 1024, 1536 ...
//Calculados como :
//   (Nr_Setor - 1) x 512. Por exemplo: Nr_Setor = 1000, resulta no endereço 0x7CE00
//-----
unsigned char escreve_MMC(uchar *buffer, uint enderecoH, uint enderecoL)
{
    unsigned int i;

    habilita_MMC();

    //CMD24 (WRITE_SINGLE_BLOCK) - escreve um único bloco de 512 bytes
    comando_MMC(0x58,enderecoH,enderecoL,0xFF);

    //retorna 1 para indicar erro do CMD24
    if (resposta_MMC(0x00)) { desabilita_MMC(); return 1; }

    SPI(0xFE);//indica o início do bloco de dados

    for(i=0;i<512;i++)//transfere os 512 bytes do buffer para o MMC
    {
        SPI(*buffer);
        buffer++;
    }

    //no final envia dois bytes sem utilidade (dummy checksum)
    SPI(0xFF);
    SPI(0xFF);

    //retorna 2 para indicar erro na escrita
    if ((SPI(0xFF) & 0x0F) != 0x05) {desabilita_MMC(); return 2; }

    //espera final de escrita
    i=0xFFFF;
    do
    {
        i--;
    } while ((SPI(0xFF)== 0x00) && i);

    //retorna 3 para indicar erro no final da escrita
    if(i==0){ desabilita_MMC(); return 3; }

    desabilita_MMC();    //desabilita o SD card
    SPI(0xFF);

    return 0; //retorna 0 para indicar sucesso na operação
}
//-----
//Lê a partir do endereço especificado (setor), sempre em blocos de 512 bytes
//(como na escrita)
//-----

```

```

unsigned char le_MMC(uchar *buffer,uint enderecoH, uint enderecoL)
{
    unsigned int i;

    habilita_MMC();

    //CMD17 (READ_SINGLE_BLOCK) - leitura de um único bloco de 512 bytes
    comando_MMC(0x51,enderecoH,enderecoL,0xFF);

    //retorna 1 para indicar erro na leitura
    if (resposta_MMC(0x00)) {desabilita_MMC(); return 1;}

    //retorna 2 para indicar erro na leitura
    if (resposta_MMC(0xFE)) {desabilita_MMC(); return 2;}

    //leitura dos 512 bytes para o vetor de memória desejado (buffer)
    for(i=0; i < 512; i++)
    {
        *buffer=SPI(0xFF); //envia 0xFF para receber o dado
        buffer++;
    }

    //no final envia dois bytes, retornam o CRC/checksum byte (sem importância no programa)
    SPI(0xFF);
    SPI(0xFF);

    desabilita_MMC();
    SPI(0xFF);

    return 0; //retorna 0 para indicar sucesso na operação
}
//-----

```

Exercícios:

14.1 – Altere o programa para o TC72 (pág. 329), apresentando a temperatura com uma resolução de 0,25 °C (fig. 14.7).

14.2 – Elaborar um programa para a leitura do sensor de temperatura MAX6675 do circuito da fig. 14.11.

O MAX6675 digitaliza o sinal proveniente de um termopar tipo K³⁸ e sua saída de dados é de 12 bits com compatibilidade SPI. Sua resolução é de 0,25°C, com medição de temperatura entre 0 e 1024°C.

³⁸ O termopar tipo K é um sensor de temperatura composto por Chromel (90% de Níquel e 10% de Cromo) e Alumel (95% de Níquel, 2% de Manganês, 2% de Alumínio e 1% de Silício). Os dois materiais são conectados e na sua junção é gerada uma pequena diferença de potencial dependendo da temperatura a que são submetidos. Fonte <http://pt.wikipedia.org/wiki/Termopar>.

Seu protocolo de comunicação é apresentado na fig. 14.12. Nela se observa que os bits mais significativos do dado são enviados primeiro. O formato do dado é apresentado na tab. 14.7.

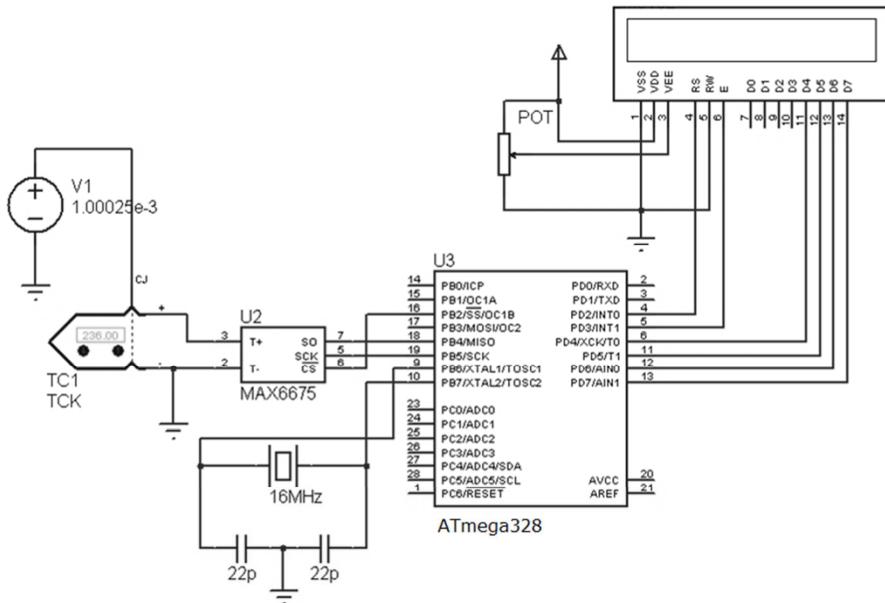


Fig. 14.11 – Usando o sensor MAX6675 (a fonte V1 é para a compensação da tensão gerada pelo termopar, utilizada para a simulação).

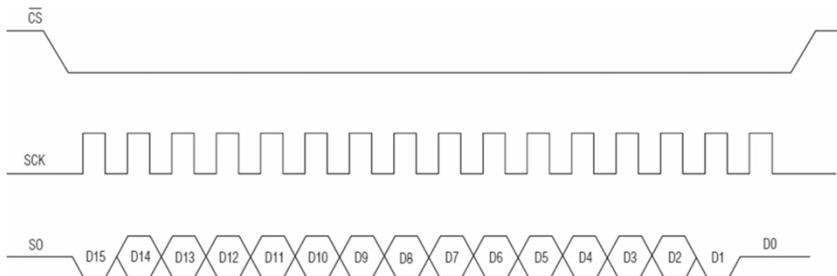


Fig. 14.12 – Protocolo de comunicação do MAX6675.

Tab. 14.7 - Formato de dados enviado pelo MAX6675.

	Bit de Sinal	Leitura de 12 bits da temperatura												Entrada do termopar	ID do dispositivo	Estado	
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3		2	1	0
	0	MSB												LSB		0	<i>Tri-State</i>

Para o emprego da SPI do ATmega, basta escrever um dado qualquer no registrador de dados da SPI (SPDR). O controle do pino \overline{CS} deve ser feito por software e os dados serão recebidos no pino MISO. O trecho de programa abaixo ilustra o processo.

```
-----
//MAX6675
...
           //configurações para uso da SPI
clr_bit(PORTB,SS); //pino SS em zero
temp_MSB =le_MAX(); //lê o byte + signific.(temp_MSB é um unsigned char)
temp_LSB =le_MAX(); //lê o byte - signific.(temp_LSB é um unsigned char)
set_bit(PORTB,SS); //pino SS em 1

valor_int = (temp_LSB>>3) | (temp_MSB<<5);      /*são 12 bits de dados, os
                                                       demais não são analisados*/
valor_int = valor_int/4; /*divide o valor por 4 para uma resolução de 1 grau,
                           ou valor_int=(temp_LSB>>5)|(temp_MSB<<3) - já divide.
                           valor_int é um unsigned int*/
...
           //aqui vai a rotina para mostrar o valor no display
//-
unsigned char le_MAX()
{
    SPDR = 0x00;           //envia um dado qualquer para receber um byte
    while(!(SPSR & (1<<SPIF))); //aguarda o envio do dado
    return SPDR;           //retorna o valor recebido
}
-----
```

14.3 – Como o MAX6675 poderia ser utilizado para o projeto de um forno para solda de componentes SMD?

14.4 – Pesquise o sistema de arquivos FAT16/FAT32 e desenvolva um programa para escrever e ler dados nesses formatos em cartões SD/MMC.

15. USART

Neste capítulo, é apresentada a USART, um periférico com características muito flexíveis e amplamente utilizado na comunicação serial dos microcontroladores com o mundo externo. O seu emprego é exemplificado na comunicação do ATmega com um computador, através de uma porta USB, na comunicação entre módulos básicos de rádio frequência e na comunicação com dispositivos *bluetooth*. No Arduino, a USART é usada principalmente para a gravação do ATmega328 através do computador.

A *Universal Synchronous and Asynchronous serial Receiver and Transmitter* é um módulo de comunicação serial com inúmeras possibilidades de configurações de trabalho, o que lhe permite ser aplicada em uma infinidade de sistemas eletrônicos. Como por exemplo, nas comunicações RS232 e RS485, que apesar de não serem mais utilizadas em computadores pessoais, são, ainda, largamente usadas em sistemas industriais de controle. A grande vantagem da USART é que muitos dispositivos eletrônicos modernos suportam seu protocolo de comunicação.

15.1 USART DO ATMEGA328

As principais características do módulo de comunicação USART do ATmega328 são:

- Operação *Full Duplex* (registradores independentes de recepção e transmissão).
- Operação síncrona ou assíncrona.
- Operação síncrona com *clock* mestre ou escravo.
- Gerador de taxa de comunicação de alta resolução (*Baud Rate Generator*).

- Suporta *frames* seriais com 5, 6, 7, 8 ou 9 bits de dados e 1 ou 2 bits de parada.
- Gerador de paridade par ou ímpar e conferência de paridade por hardware.
- Detecção de colisão de dados e erros de *frames*.
- Filtro para ruído, incluindo bit de inicio falso e filtro digital passa-baixa.
- Três fontes separadas de interrupção (transmissão completa, recepção completa e esvaziamento do registrador de dados).
- Modo de comunicação assíncrono com velocidade duplicável.
- Pode ser utilizada como interface SPI mestre.

Para gerar a taxa de comunicação no modo mestre, é empregado o registrador UBRR0 (USART *Baud Rate Register 0*). Um contador decrescente trabalhando na velocidade de *clock* da CPU é carregado com o valor de UBRR0 cada vez que chega a zero ou quando o UBRR0 é escrito. Assim, um pulso de *clock* é gerado cada vez que esse contador zera, determinando a taxa de comunicação (*baud rate*). O transmissor dividirá o *clock* de *baud rate* por 2, 8 ou 16, de acordo com o modo programado. A taxa de transmissão de saída é usada diretamente pela unidade de recepção e recuperação de dados. Na tab. 15.1, são apresentadas as equações para o cálculo da taxa de comunicação (bits por segundo - bps) e para o cálculo do valor de UBRR0 para cada modo de operação usando a fonte de *clock* interna (para valores previamente calculados, ver a tab. C1 do apêndice C).

Tab. 15.1 – Equações para o cálculo do registrador UBRR0 da taxa de transmissão.

Modo de operação	Equação para o cálculo da taxa de transmissão	Equação para o cálculo do valor de UBRR0
Modo Normal Assíncrono (U2X0 = 0)	$TAXA = \frac{f_{osc}}{16(UBRR0 + 1)}$	$UBRR0 = \frac{f_{osc}}{16.TAXA} - 1$
Modo de Velocidade Dupla Assíncrono (U2X0 = 1)	$TAXA = \frac{f_{osc}}{8(UBRR0 + 1)}$	$UBRR0 = \frac{f_{osc}}{8.TAXA} - 1$
Modo Mestre Síncrono	$TAXA = \frac{f_{osc}}{2(UBRR0 + 1)}$	$UBRR0 = \frac{f_{osc}}{2.TAXA} - 1$

Para duplicar a taxa de comunicação no modo assíncrono, basta ativar o bit U2X0 do registrador UCSR0A. Esse bit deve ser colocado em zero na operação síncrona.

Emprega-se um sinal externo de *clock* no pino XCK para os modos síncronos de recepção escravo. A frequência máxima desse sinal está limitada pelo tempo de resposta da CPU, devendo seguir a seguinte condição:

$$f_{XCK} < \frac{f_{osc}}{4} \quad (15.1)$$

A frequência do sistema (f_{osc}) depende da estabilidade da fonte de *clock*. É recomendado usar alguma margem de segurança na eq. 15.1 para evitar uma possível perda de dados.

Quando o modo síncrono é usado (UMSEL0=1), o pino XCK será empregado como *clock* de entrada (escravo) ou saída (mestre). O princípio básico é que o dado de entrada (no pino RXD) é amostrado na borda oposta do XCK quando a borda do dado de saída é alterada (pino TXD). O bit UCPOLO do registrador UCRS0C seleciona qual borda de XCK é usada para a amostragem do dado e qual é usada para a mudança do dado. Na fig. 15.1, o efeito do bit UCPOLO é apresentado.

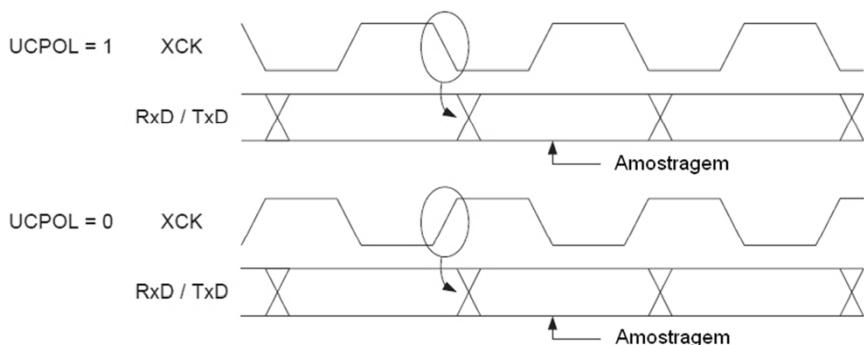


Fig. 15.1 – Efeito do bit UCPOLO na amostragem dos dados.

O grupo de bits é transmitido/recebido em um bloco (*frame*) composto pelos bits de dados, bits de sincronização (bits de início e parada) e, opcionalmente, por um bit de paridade para a conferência de erro. A USART aceita várias combinações possíveis de formato de dados:

- Um bit de início.
- 5, 6, 7, 8 ou 9 bits de dados.
- Bit de paridade par, ímpar ou nenhum.
- Um ou dois bits de parada.

Um *frame* inicia com um bit de início, seguido pelo bit menos significativo (LSB). Seguem os outros bits de dados, num total de até 9 bits. O *frame* termina com o bit mais significativo (MSB). Se habilitado, o bit de paridade é inserido após os bits de dados, antes dos bits de parada. Após a transmissão completa dos bits, pode-se seguir outra transmissão ou aguardar-se nova transmissão. O formato do número total de bits (*frame*) é ilustrado na fig. 15.2.

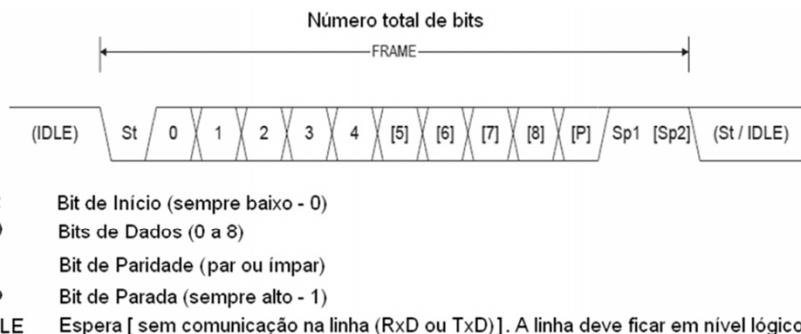


Fig. 15.2 – Formato do *frame* da USART.

O formato do *frame* é definido pelos bits UCSZ0:0, UPM01:0 e USBS0 nos registradores UCSR0B e UCSR0C. O transmissor e o receptor usam a mesma configuração.

Antes de qualquer comunicação, a USART deve ser inicializada. Os detalhes de configuração necessitam ser ajustados nos registradores específicos, detalhados posteriormente. A seguir, são apresentados exemplos para trabalho com a USART a partir de códigos extraídos do manual do fabricante.

```

//=====
//      INICIALIZANDO A USART
//=====

#define FOSC    1843200      //Frequência de trabalho da CPU
#define BAUD    9600
#define MYUBRR FOSC/16/BAUD-1
//-----
void main(void)
{
    ...
    USART_Init(MYUBRR);
    ...
}

void USART_Init(unsigned int ubrr)
{
    UBRR0H = (unsigned char)(ubrr>>8); //Ajusta a taxa de transmissão
    UBRR0L = (unsigned char)ubrr;
    UCSR0B = (1<<RXEN0)|(1<<TXEN0); //Habilita o transmissor e o receptor
    UCSR0C = (1<<USBS0)|(3<<UCSZ00); //Ajusta o formato do frame:
                                         //8 bits de dados e 2 de parada
}
//=====

//=====
//      ENVIANDO FRAMES COM 5 A 8 BITS
//=====

void USART_Transmit(unsigned char data)
{
    while(!( UCSR0A & (1<<UDRE0))); //Espera a limpeza do registr. de transmissão
    UDR0 = data;                      //Coloca o dado no registrador e o envia
}

//=====

//=====
//      ENVIANDO FRAMES COM 9 BITS
//=====

void USART_Transmit(unsigned int data)
{
    while(!(UCSR0A &(1<<UDRE0))); //Espera a limpeza do registr. de transmissão

    UCSR0B &= ~(1<<TXB80);        //Copia o 9º bit para o TXB8

    if(data & 0x0100)
        UCSR0B |= (1<<TXB80);

    UDR0 = data;                  //Coloca o dado no registrador e o envia
}
//=====
```

```

//=====
//      RECEBENDO FRAMES COM 5 A 8 BITS          //
//=====                                         //
unsigned char USART_Receive(void)
{
    while(!(UCSR0A & (1<<RXC0))); //Espera o dado ser recebido
    return UDR0;                      //Lê o dado recebido e retorna
}
//=====

//=====                                         //
//      RECEBENDO FRAMES COM 9 BITS           //
//=====                                         //
unsigned int USART_Receive(void)
{
    unsigned char status, resh, resl; //Espera o dado ser recebido
    while(!(UCSR0A & (1<<RXC0))); //Obtém o status do 9º bit, então, o dado do registr.
    status = UCSR0A;
    resh = UCSR0B;
    resl = UDR0;

    if(status & (1<<FE0)|(1<<DOR0)|(1<<UPE0)) //Se ocorrer um erro retorna -1
        return -1;

    resh = (resh >> 1) & 0x01;                  //Filtrar o 9º bit, então, retorna
    return ((resh << 8) | resl);
}
//=====

//=====                                         //
// LIMPANDO O REGISTRADOR DE ENTRADA (quando ocorre um erro p. ex.)          //
//=====                                         //
void USART_Flush(void)
{
    unsigned char dummy;
    while(UCSR0A & (1<<RXC0)) dummy = UDR0;
}
//=====

```

A faixa de operação do receptor é dependente do descasamento entre a taxa de comunicação do transmissor e a sua taxa de comunicação, gerada internamente (*baud rate*). Se o transmissor estiver enviando bits muito rapidamente, ou muito devagar, ou ainda, se o *clock* gerado internamente não tiver a frequência base similar a do transmissor, o receptor não será capaz de sincronizar os *frames* relativos ao bit de início. Isso gerará um erro na taxa de recepção, que, segundo a Atmel, deve estar entre $\pm 1\%$ e $\pm 3\%$ (ver manual do componente, tabelas detalhadas são apresentadas no apêndice C). O erro é calculado por:

$$Erro [\%] = \frac{\text{Taxa de Transmissão Aproximada}}{\text{Taxa de Transmissão Ideal}} \cdot 100\% \quad (15.2)$$

TRANSMISSÃO DE DADOS

Uma transmissão é iniciada ao se escrever o dado a ser transmitido no registrador de I/O da USART – UDR0. O dado é transferido de UDR0 para o registrador de deslocamento de transmissão quando ele está no estado *idle* (ocioso) ou imediatamente após o último bit de parada do frame anterior ter sido deslocado para a saída.

Se o registrador de deslocamento do transmissor está vazio, o dado é transferido do registrador UDR0 para o registrador de deslocamento. Neste momento, o bit UDRE0 (*USART Data Register Empty 0*) no registrador UCSR0A é posto em 1 lógico. Quando esse bit está em 1 lógico, a USART está pronta para receber o próximo caractere.

No ciclo de *clock* da taxa de comunicação seguinte à operação de transferência para o registrador de deslocamento, o bit de início é deslocado para o pino TXD, seguindo o dado com o LSB primeiro. Quando o último bit de parada é deslocado para a saída, o registrador de deslocamento é carregado se algum novo dado foi escrito no UDR0 durante a transmissão. Durante a carga, UDRE0 é posto em 1 lógico. Se não há nenhum dado novo no registrador UDR0 para ser enviado quando o bit de parada é deslocado, o *flag* UDRE0 permanecerá em 1 lógico até UDR0 ser escrito novamente. Se nenhum dado novo foi escrito e o bit de parada estiver presente em TXD pelo tempo de um bit, o *flag* de transmissão completa (TXC0) em UCSR0A é posto em 1.

O bit TXEN0 no registrador UCSR0B habilita o transmissor da USART quando em 1 lógico. Se esse bit é posto em 0 lógico, o pino PD1 pode ser usado para I/O de dados. Quando TXEN0 é posto em 1 lógico, o transmissor da USART é conectado a PD1, o qual é forçado a ser um pino de saída independente da configuração do bit DDD1 em DDRD.

RECEPÇÃO DE DADOS

O receptor inicia a recepção de dados quando um bit de início válido é detectado. Os bits seguintes são amostrados e deslocados para o registrador de deslocamento de recepção até o primeiro bit de parada de um *frame* ser recebido. Neste momento, o conteúdo do registrador de deslocamento é movido para o registrador de recepção e o bit RXC0 em UCSR0A é posto em 1 lógico. O dado recebido pode ser lido a partir do registrador UDR0.

Quando o bit RXEN0 no registrador UCSR0A está em 0 lógico, a recepção está desabilitada. Isso significa que o terminal PD0 pode ser usado como um pino de I/O. Quando RXEN0 está em 1 lógico, o receptor da USART está conectado a PD0, o que o força a ser uma entrada, independente da configuração do bit DDD0 em DDRD. Quando PD0 é forçado como entrada pela USART, o PORTD pode ainda ser usado para controlar o resistor de *pull-up* do pino.

REGISTRADORES DA USART

UDR0 – USART I/O Data Register:

Bit	7	6	5	4	3	2	1	0
UDR0 (leitura)					RXB[7:0]			
UDR0 (escrita)					TXB[7:0]			
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Os registradores de recebimento e envio de dados compartilham o mesmo endereço lógico. Entretanto, um leitura é feita no UDR0 de leitura e uma escrita, no UDR0 de escrita, o hardware se encarrega da distinção. Para frames com 5, 6 ou 7 bits, os bits não utilizados serão ignorados e na recepção colocados em zero. O UDR0 só deve ser escrito quando o bit UDRE0 do registrador UCSR0A estiver ativo.

UCSROA – USART Control and Status Register A:

Bit	7	6	5	4	3	2	1	0
UCSROA	RXC0	TXC0	UDRE0	FEO	DOR0	UPE0	U2X0	MPCMO
Lê/Escr.	L	L/E	L	L	L	L	L/E	L/E
Valor Inicial	0	0	1	0	0	0	0	0

Bit 7 – RXC0 – USART Receive Complete

Este bit é posto em 1 lógico quando um caractere recebido é transferido do registrador de deslocamento de recepção para o registrador de recepção. O bit RXC0 é posto em 0 lógico na leitura de UDR0.

Bit 6 – TXC0 – USART Transmit Complete

Este bit é posto em 1 lógico quando um caractere completo (incluindo o bit de parada) no registrador de deslocamento de transmissão for transferido e nenhum dado foi escrito em UDR0. Este bit é especialmente útil em interfaces de comunicação *half-duplex*, onde um aplicativo de transmissão deve entrar no modo de recepção e liberar o barramento de comunicação imediatamente após completar a transmissão. O bit TXC0 é posto em 0 lógico pelo hardware ao executar o vetor correspondente de tratamento de interrupção. Alternativamente, o bit TCX0 é limpo pela escrita de 1 lógico.

Bit 5 – UDRE0 – USART Data Register Empty

Este bit é posto em 1 lógico quando um caractere escrito no UDR0 é transferido para o registrador de deslocamento de transmissão. Um lógico neste bit indica que o transmissor está pronto para receber um novo caractere para transmissão. O bit UDRE0 é posto em 0 lógico na leitura de UDR0. Quando uma interrupção por USART *Data Register Empty* deve escrever em UDR0 a fim de limpar UDRE0, se não o fizer, uma nova interrupção ocorrerá a cada vez que a rotina de interrupção terminar. UDRE0 é posto em 1 lógico durante a inicialização para indicar que o transmissor está pronto.

Bit 4 – FEO – Frame Error

Indica se existe um erro no *frame* recebido. Este bit deve sempre ser zerado quando se escreve no registrador UCSROA.

Bit 3 – DOR0 – Data OverRun

Ocorre quando o registrador de entrada está cheio, não foi lido e um novo bit de início é detectado. Este bit deve sempre ser zerado quando se escreve no registrador UCSROA.

Bit 2 – UPE0 – USART Parity Error

Indica se existe um erro de paridade no dado recebido e fica ativo até UDR0 ser lido. Este bit deve sempre ser zerado quando se escreve no registrador UCSROA.

Bit 1 – U2X0 – Double the USART transmission speed

Este bit só tem efeito no modo de operação assíncrona.

Bit 0 – MPCMO – Multi-processor Communication Mode

Habilita a comunicação com vários processadores. Quando ativo, todos os *frames* recebidos serão ignorados se não contiverem uma informação de endereço.

UCSR0B – USART Control and Status Register B:

Bit	7	6	5	4	3	2	1	0
UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXENO	TXENO	UCSZ02	RXB80	TXB80
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L	L/E

Valor Inicial

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Bit 7 – RXCIE0 – RX Complete Interrupt Enable

Este bit habilita a interrupção por recepção de dados completa (bit RXCO). Uma interrupção de recepção será gerada somente se o bit RXCIE0, o bit I (SREG) e o bit RXCO estiverem ativos.

Bit 6 – TXCIE0 – TX Complete Interrupt Enable

Este bit habilita a interrupção por transmissão de dados completa (bit TXCO). Uma interrupção de transmissão será gerada somente se o bit TXCIE0, o bit I (SREG) e o bit TXCO estiverem ativos.

Bit 5 – UDRIE0 – USART Data Register Empty Interrupt Enable

Este bit habilita a interrupção por registrador de dados vazio (bit UDRE0). Uma interrupção por registrador de dados vazio será gerada somente se o bit UDRIE0, o bit I (SREG) e o bit UDRE0 estiverem ativos.

Bit 4 – RXENO – Receiver Enable

Este bit habilita a recepção da USART. O receptor irá alterar a operação normal do pino RXD. Desabilitando o receptor, ocorrerá o esvaziamento do registrador de entrada, invalidando os bits FE0, DOR0 e UPE0.

Bit 3 – TXENO – Transmitter Enable

Este bit habilita a transmissão da USART. O transmissor irá alterar a operação normal do pino TXD. A desabilitação do transmissor só terá efeito após as transmissões pendentes serem completadas.

Bit 2 – UCSZ02 – Character Size

Este bit, combinado com os bits UCSZ01:0 do registrador UCSR0C, ajusta o número de bits de dados do *frame*.

Bit 1 – RXB80 – Receive Data Bit 8

É o nono bit de dados recebidos quando o *frame* for de 9 bits. Deve ser lido antes dos outros bits do UDR0.

Bit 0 – TXB80 – Transmit Data Bit 8

É o nono bit de dados a ser transmitido quando o *frame* for de 9 bits. Deve ser escrito antes dos outros bits no UDR0.

UCSROC - USART Control and Status Register C:

Bit	7	6	5	4	3	2	1	0
UCSROC	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOLO
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	1	1	0

Bit 7:6 – UMSEL01:0 – USART Mode Select

Estes bits selecionam o modo de operação da USART, conforme tab. 15.2.

Tab. 15.2 – Ajuste dos bits UMSEL01:0 para o modo de operação da USART.

UMSEL01	UMSEL00	Modo de operação
0	0	assíncrono
0	1	síncrono
1	0	reservado
1	1	SPI mestre

Bits 5:4 – UPM01:0 – Parity Mode

Estes bits habilitam e ajustam o gerador de paridade e de conferência. Se habilitado, o transmissor irá gerar e enviar automaticamente o bit de paridade em cada *frame*; o receptor irá gerar o valor de paridade para comparação. Se uma desigualdade for detectada, o bit UPE0 torna-se ativo. Na tab. 15.3, são apresentadas as possíveis configurações para os bits UPM01:0.

Tab. 15.3 – Bits UPM01:0.

UPM01	UPM00	Modo de Paridade
0	0	Desabilitado
0	1	Reservado
1	0	Habilitado, paridade par
1	1	Habilitado, paridade ímpar

Bit 3 – USBS0 – Stop Bit Select

Este bit seleciona o número de bits de parada a serem inseridos pelo transmissor (USBS0=0, 1 bit de parada; USBS0=1, 2 bits de parada).

Bits 2:1 – UCSZ01:0 – Character Size

Estes bits, combinados com o bit UCSZ02 do registrador UCSR0B, ajustam o número de bits de dados no *frame* do transmissor e receptor, conforme tab. 15.4.

Tab. 15.4 – Ajuste dos bits UCSZ01:0.

UCSZ02	UCSZ01	UCSZ00	Tamanho do Caractere
0	0	0	5 bits
0	0	1	6 bits
0	1	0	7 bits
0	1	1	8 bits
1	0	0	reservado
1	0	1	reservado
1	1	0	reservado
1	1	1	9 bits

Bit 0 – UCPOLO – Clock Polarity

Este bit é usado somente no modo síncrono. Deve ser zero quando o modo assíncrono é usado. Ele ajusta o sinal de *clock* (XCK) para amostragem e saída de dados, conforme tab. 15.5.

Tab. 15.5 – Ajustando a polaridade do *clock*.

UCPOL0	Mudança do Dado Transmitido (saída do pino TxD0)	Amostragem do Dado Recebido (entrada do pino RxD0)
0	borda de subida de XCK	borda de descida de XCK
1	borda de descida de XCK	borda de subida de XCK

UBRR0L e UBRR0H - USART Baud Rate Register

Bit	15	14	13	12	11	10	9	8				
UBRR0H	-	-	-	-	UBRR[11:8]							
UBRR0L	UBRR[7:0]											
Bit	7	6	5	4	3	2	1	0				
Lê/Escrive	L	L	L	L	L/E	L/E	L/E	L/E				
	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E				
Valor Inicial	0	0	0	0	0	0	0	0				
	0	0	0	0	0	0	0	0				

Bits 15:12 – Reservado

Devem ser zero quando se escreve em UBRR0H.

Bits 11:0 – UBRR011:0: USART Baud Rate Register

Este é o registrador de 12 bits que contém o valor da taxa de comunicação. Qualquer transmissão em andamento será corrompida se houver mudança desse valor. Qualquer escrita atualiza imediatamente a taxa de comunicação, ver a tab. 15.1.

15.2 USART NO MODO SPI MESTRE

A USART do ATmega328 também pode ser empregada como interface SPI mestre com as seguintes características:

- Operação *Full Duplex*, transferência síncrona de dados.
- Suporta todos os modos de operação da SPI: 0, 1, 2 e 3.
- Escolha do primeiro bit para transmissão (LSB ou MSB).
- Gerador de *clock* de alta resolução.
- Frequência máxima igual à metade da CPU.
- Permite interrupções.

Nas tabs. 15.6 e 15.7, são apresentadas as equações para o cálculo da taxa de transmissão e as configurações dos bits para os diferentes modos de operação da SPI.

Tab. 15.6 – Equações para calcular a taxa de comunicação da USART no modo SPI mestre.

Equação para o cálculo da taxa de transmissão	Equação para o cálculo do valor de UBRR0
$TAXA = \frac{f_{osc}}{2(UBRR0 + 1)}$	$UBRR0 = \frac{f_{osc}}{2.TAXA} - 1$

Tab. 15.7 – Bits de configuração para os modos SPI.

UCPOL0	UCPHAO	Borda do SCK	Borda do SCK	MODO SPI
0	0	Amostragem (subida)	Ajuste (descida)	0
0	1	Ajuste (subida)	Amostragem (descida)	1
1	0	Amostragem (descida)	Ajuste (subida)	2
1	1	Ajuste (descida)	Amostragem (subida)	3

Funções exemplo são apresentadas a seguir, conforme manual do fabricante.

```

//-----
void USART_Init( unsigned int baud )
{
    UBRR0 = 0;

    DDRD |= (1<<PD4);      //pino XCK como saída, habilita o modo mestre

    //Modo MSPI e SPI no modo 0
    UCSR0C = (1<<UMSEL01)|(1<<UMSEL00)|(0<<UCPHA0)|(0<<UCPOL0);
    UCSR0B = (1<<RXEN0)|(1<<TXEN0);      //habilita a transmissão e recepção

    // Ajuste da taxa de transmissão
    // IMPORTANTE: A taxa de transmissão deve ser ajustada após a habilitação
    UBRR0 = baud;      //a variável baud deve conter o valor da taxa
}

//-----
unsigned char USART_TXRX(unsigned char dado)
{
    while(!(UCSR0A & (1<<UDRE0)) );      //espera a transmissão ser completada

    UDR0 = dado;                      //envia o dado

    while(!(UCSR0A & (1<<RXC0)) );      //espera o dado ser recebido

    return UDR0;                      //retorna o dado recebido
}
//-----

```

15.3 COMUNICAÇÃO ENTRE O MICROCONTROLADOR E UM COMPUTADOR

Quem começa os seus passos como programador, logo se depara com algum programa para a comunicação entre um computador e o mundo externo. Antigamente, empregavam-se as portas paralela ou serial disponíveis no hardware (onde se colocavam a impressora e o mouse, por exemplo). Na atualidade, os computadores não dispõem mais dessas portas, o domínio é das portas USB (*Universal Serial Bus*). Entretanto, isso não impede que dispositivos que usem porta paralela ou serial sejam conectados ao computador. Uma solução é o emprego de cabos conversores que podem ser ligados diretamente às portas USB. Na fig. 15.3, são

apresentados os terminais de um cabo conversor serial/USB³⁹ (COMTAC®) empregado na comunicação RS232.



Fig. 15.3 – Terminais de um cabo para ligar um dispositivo com comunicação RS232 a um computador com entrada USB.

O fabricante do cabo conversor fornece o programa de instalação do seu dispositivo no computador (*driver*). Esse programa instala uma porta serial virtual no computador para ser utilizada pela USB. Após a instalação do *driver*, é necessário um programa para realizar a comunicação. Esse programa pode ser desenvolvido pelo usuário ou, em aplicações simples, pode-se empregar um programa gratuito. Nas versões anteriores do sistema operacional Windows XP, era disponibilizada uma ferramenta com essa função, chamada de hiperterminal. Em versões posteriores do Windows, onde essa ferramenta não existe, é necessário utilizar outro programa.

Neste trabalho, foi empregado o programa gratuito Hércules da HW Group (www.hw-group.com). Na fig. 15.4, é apresentada a janela de configuração para a comunicação serial, onde se define qual porta virtual

³⁹ O desempenho conseguido com um conversor serial/USB é inferior ao da USB isoladamente. Literalmente, a comunicação não pode ser denominada USB, pois o protocolo e as taxas de transmissão são diferentes.

(criado pelo programa do cabo conversor) será empregada (neste caso, a COM3), qual a taxa de transmissão (9600 bps) e as outras características da comunicação que se deseja realizar.

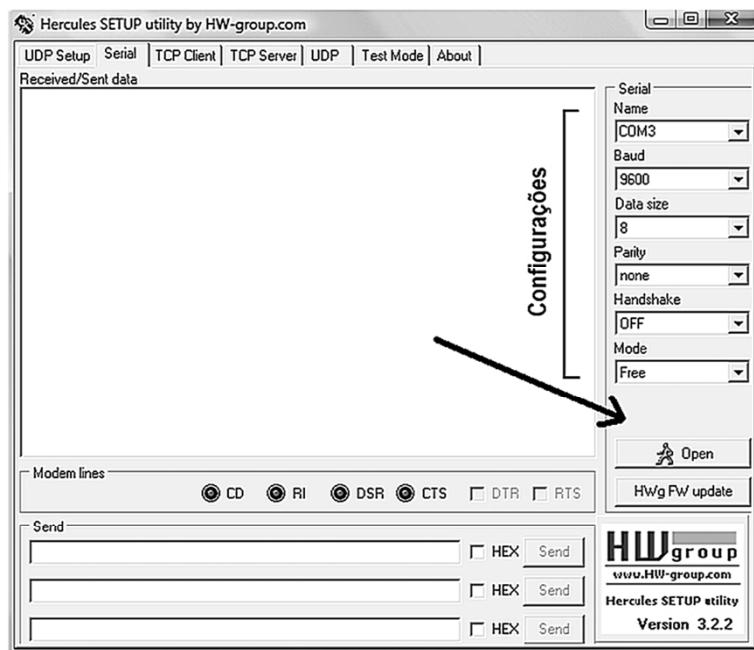


Fig. 15.4 – Configurando o programa Hercules para uma comunicação RS232.

Após todos os programas instalados e configurados, a comunicação com o dispositivo externo desejado pode ser feita. Todavia, os níveis lógicos que o circuito do cabo conversor entende, são os níveis de tensão do protocolo RS232 e, portanto, deve ser utilizado um circuito para adequar os níveis de tensão do microcontrolador aos níveis compreendidos pelo conversor, como por exemplo, o CI MAX232, como apresentado na fig. 15.5 (a alimentação desse CI é a mesma do microcontrolador). O circuito externo deve dispor do conector DB-9 fêmea para a conexão do cabo.

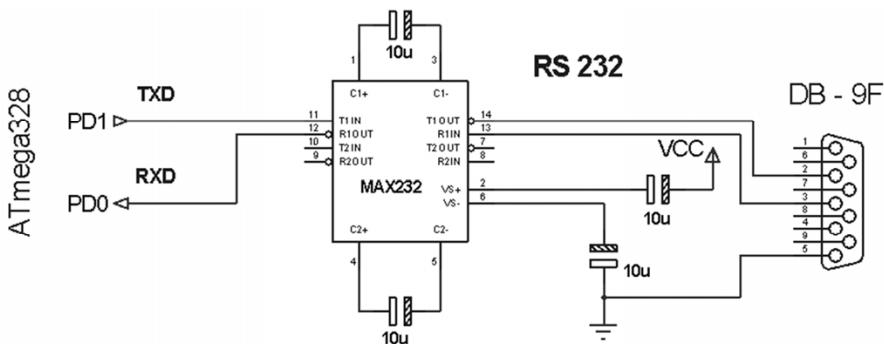


Fig. 15.5 – Circuito para adequação da tensão proveniente de um microcontrolador para a comunicação RS232.

O uso do cabo conversor é uma solução barata e de fácil aquisição. O seu porém, é a necessidade de utilização do MAX232 ou equivalente. Outra possibilidade, é o uso de um circuito integrado dedicado para realizar a comunicação diretamente com os níveis de tensão do circuito microcontrolado, como por exemplo, o FT232. Circuitos conversores desse tipo podem ser encontrados prontos para uso, mas o seu custo geralmente é superior ao dos cabos conversores.

O Arduino Duemilenove emprega um FT232 e o Arduino Uno utiliza um ATmega8U2 com USB para a comunicação com um computador (ver a fig. 3.2). Dessa forma, a plataforma Arduino pode ser ligada diretamente ao computador com o uso de apenas um cabo USB. Quando o Arduino é conectado ao computador, o *driver* do dispositivo conversor deve ser instalado e uma porta COM será associada automaticamente a ele.

Com a plataforma adequadamente instalada, basta escrever o código para o microcontrolador e conectá-lo ao computador. Na fig. 15.6, é apresentada a comunicação realizada entre o computador e o Arduino Uno. O microcontrolador transmite a mensagem inicial: “Transmitindo primeira mensagem para o computador! Digite agora – Para sair <*>”. O programa microcontrolado retorna qualquer caractere recebido com os símbolos “->” seguidos do caractere. Também é possível enviar uma palavra inteira pelo

programa de comunicação, a qual aparecerá na janela do programa com a cor rosa.

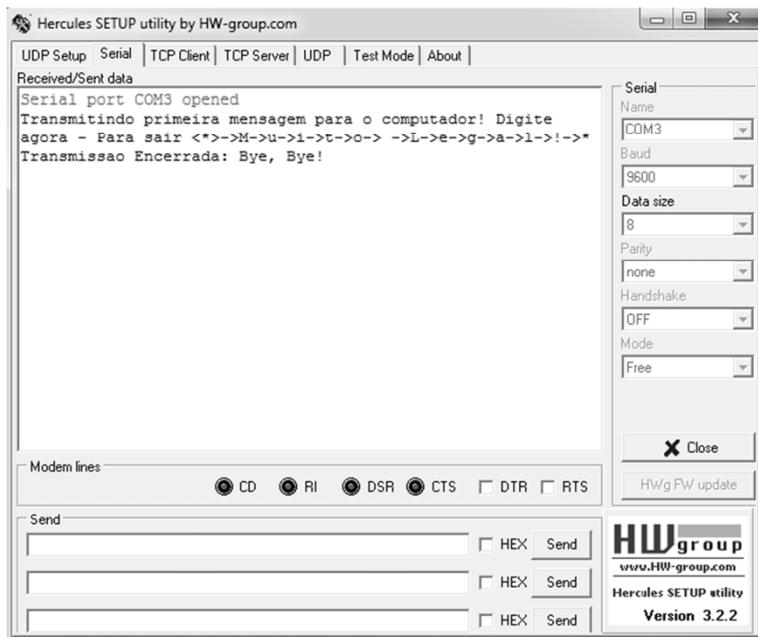


Fig. 15.6 – Janela do programa teste de comunicação entre um computador e Arduino Uno.

O programa utilizado para a comunicação no exemplo acima é apresentado a seguir. O programa foi desenvolvido para esperar o recebimento e o envio de um caractere por vez. Quando essa espera não for viável, a interrupção da USART deve ser empregada.

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL //define a frequência do microcontrolador - 16MHz

#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
```

```

//Definições de macros para o trabalho com bits
#define    set_bit(y,bit)  (y|=(1<<bit))
#define    clr_bit(y,bit)  (y&=~(1<<bit))
#define    cpl_bit(y,bit)  (y^=(1<<bit))
#define    tst_bit(y,bit)  (y&(1<<bit))

#endif

```

USART_PC.c (programa principal)

```

//=====================================================================
//          COMUNICAÇÃO SERIAL ENTRE O ARDUINO E UM COMPUTADOR      //
//===================================================================== //
#include "def_principais.h"
#include "USART.h"

const char primeira_msg[] PROGMEM = "Transmitindo primeira mensagem para o computador!
                                         Digite agora - Para sair *>\n\0";
const char segunda_msg[] PROGMEM = "\nTransmissão Encerrada: Bye, Bye!\n\0";
int main()
{
    unsigned char dado_recebido;

    USART_Inic(MYUBRR);

    escreve_USART_Flash(primeira_msg);

    do
    {
        dado_recebido= USART_Recebe();           //recebe caractere
        USART_Transmite('-');
        USART_Transmite('>');
        USART_Transmite(dado_recebido);         //envia o caractere recebido
    }while(dado_recebido!='*');

    escreve_USART_Flash(segunda_msg);

    while(1);//laço infinito
}
//-----

```

USART.h (arquivo de cabeçalho do USART.c)

```

#ifndef _USART_H
#define _USART_H

#include "def_principais.h"

#define BAUD    9600      //taxa de 9600 bps
#define MYUBRR F_CPU/16/BAUD-1
#define tam_vetor  5      //número de dígitos individuais para a conversão por ident_num()
#define conv_ascii 48     /*48 se ident_num() deve retornar um número no formato ASCII
                           (0 para formato normal)*/

void USART_Inic(unsigned int ubbr0);
void USART_Transmite(unsigned char dado);
unsigned char USART_Recebe();
void escreve_USART(char *c);

```

```

void escreve_USART_Flash(const char *c);
void ident_num(unsigned int valor, unsigned char *disp);

#endif

```

USART.c (arquivo com as funções para o trabalho com a USART)

```

#include "USART.h"
//-----
void USART_Inic(unsigned int ubrr0)
{
    UBRR0H = (unsigned char)(ubrr0>>8); //Ajusta a taxa de transmissão
    UBRR0L = (unsigned char)ubrr0;

    UCSR0A = 0;//desabilitar velocidade dupla (no Arduino é habilitado por padrão)
    UCSR0B = (1<<RXEN0)|(1<<TXEN0); //Habilita a transmissão e a recepção
    UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);/*modo assíncrono, 8 bits de dados, 1 bit de
                                         parada, sem paridade*/
}
//-----
void USART_Transmite(unsigned char dado)
{
    while (!(UCSR0A & (1<<UDRE0)) ); //espera o dado ser enviado
    UDR0 = dado; //envia o dado
}
//-----
unsigned char USART_Recebe()
{
    while (!(UCSR0A & (1<<RXC0))); //espera o dado ser recebido
    return UDR0; //retorna o dado recebido
}
//-----
void escreve_USART(char *c) //escreve String (RAM)
{
    for (; *c!=0;c++) USART_Transmite(*c);
}
//-----
void escreve_USART_Flash(const char *c) //escreve String (Flash)
{
    for (;pgm_read_byte(&(*c))!=0;c++) USART_Transmite(pgm_read_byte(&(*c)));
}
//-----
//Conversão de um número em seus dígitos individuais
//-----
void ident_num(unsigned int valor, unsigned char *disp)
{
    unsigned char n;

    for(n=0; n<tam_vetor; n++)
        disp[n] = 0 + conv_ascii; //limpa vetor para armazenagem dos dígitos
    do
    {
        *disp = (valor%10) + conv_ascii; //pega o resto da divisão por 10
        valor /=10; //pega o inteiro da divisão por 10
        disp++;
    }
    }while (valor!=0);
}
//-----

```

Exercícios:

15.1 – Elaborar um programa para realizar uma comunicação serial (RS232) entre o AVR e um computador, conforme exemplo da fig. 15.7. O programa deve escrever “TESTE SERIAL” na primeira linha do LCD e mandar uma mensagem ao terminal de recepção. Os caracteres digitados nesse terminal devem ser escritos na segunda linha do LCD. Quando a escrita estiver completa, deve ser apagada e reiniciada. Devem ser empregados 2400 bps, 8 bits de dados, sem paridade e 1 bit de parada.

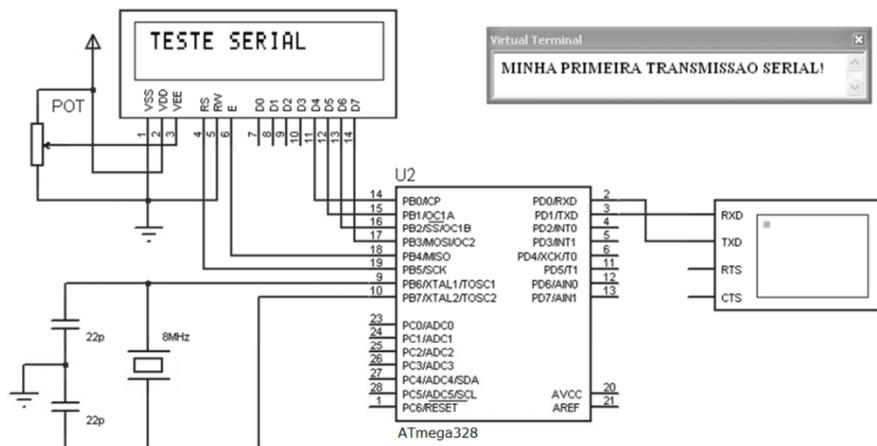


Fig. 15.7 – Comunicação serial: possível simulação no Proteus.

15.2 – Faça um programa para controlar, através do computador, 4 LEDs ligados aos pinos PB0:4 do ATmega328. Imprima na tela do hiperterminal uma mensagem semelhante a da fig. 15.8, crie sua própria imagem.

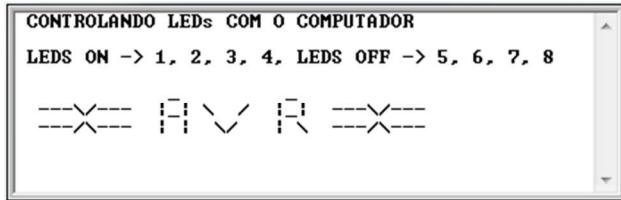


Fig. 15.8 – Mensagem recebida no computador.

A ideia é: quando o número 1 for pressionado, o LED ligado ao pino PB0 é acionado, quando o número 5 ele será desligado (comando similares para os demais LEDs).

Para gerar a mensagem da fig. 15.8 é necessário escrever cada linha individualmente na programação, tal como:

```
const char msg1 [] PROGMEM = " CONTROLANDO LEDs COM O COMPUTADOR\0";
const char msg2 [] PROGMEM = " LEDS ON -> 1, 2, 3, 4, LEDS OFF -> 5, 6, 7, 8\0";
const char msg3 [] PROGMEM = " \0";
const char msg4 [] PROGMEM = " ---\r\n--- | \r\n| / | \r\n| ---\r\n---\r\n\0";
const char msg5 [] PROGMEM = " ---/\r\n--- | \r\n| \r\n| \r\n| ---/\r\n---\r\n\0";
```

Obs.: caso seja feita uma simulação no Proteus, para o deslocamento do cursor para uma nova linha é necessário enviar o valor 0x0D para o seu terminal virtual.

*A programação exigida por este exercício é muito utilizada para a automação de sistemas, onde é necessário enviar e receber dados ou comandos de um microcontrolador.

15.3 – Altere o programa exemplo apresentado anteriormente de tal forma que ele realize a comunicação com o computador empregando as interrupções da USART.

15.4 MÓDULOS BÁSICOS DE RF

Podem ser entendidos como módulos básicos de rádio frequência (RF) os dispositivos eletrônicos constituídos por um transmissor e um receptor de ondas de rádio, compreendidas nas frequências de 315 MHz ou 433 MHz (para aplicação Industrial, Científica e Médica – ISM), que empregam principalmente a modulação ASK (*Amplitude Shift Keying*).

Antigamente, os módulos básicos de RF eram muito utilizados em eletrônica digital e, apesar de suas limitações, seu legado se estende até os dias atuais devido a facilidade de uso e ao baixo custo. Um modelo comum é apresentado na fig. 15.9. Os módulos são constituídos por dois circuitos independentes, um transmissor e um receptor. A comunicação é exclusivamente unilateral, apenas um circuito transmite e apenas um recebe, o que limita suas aplicações a controles remotos.



Fig. 15.9 –Módulos básicos de RF.

O transmissor modula digitalmente a amplitude de transmissão de acordo com o nível lógico do sinal proveniente de um circuito de controle. Essa é a modulação ASK, onde os zeros e uns modificam a amplitude do sinal transmitido. No caso do módulo básico de RF, o transmissor é ligado e desligado de acordo com os zeros e uns do dado a ser transmitido (também chamada modulação OOK – *On/Off Keying*), ver a fig. 15.10. O módulo receptor capta o sinal emitido pelo transmissor e realiza a demodulação, transformando o sinal analógico recebido nos zeros e uns provenientes da modulação.

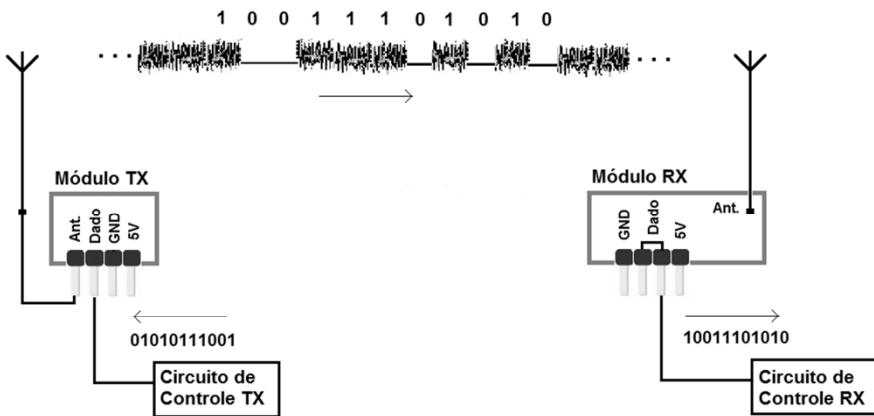


Fig. 15.10 – Diagrama de trabalho dos módulos básicos de RF.

A modulação ASK é fácil de modular e demodular e opera em uma pequena faixa de frequência (existirão algumas frequências harmônicas próximas da frequência de transmissão). Seu principal problema é a baixa imunidade a ruídos.

Uma vez ligado, o módulo transmissor transmite continuamente o sinal de rádio frequência, o que não é adequado para aplicações que utilizam baterias. Entretanto, o circuito de controle pode ligar e desligar o transmissor quando necessário.

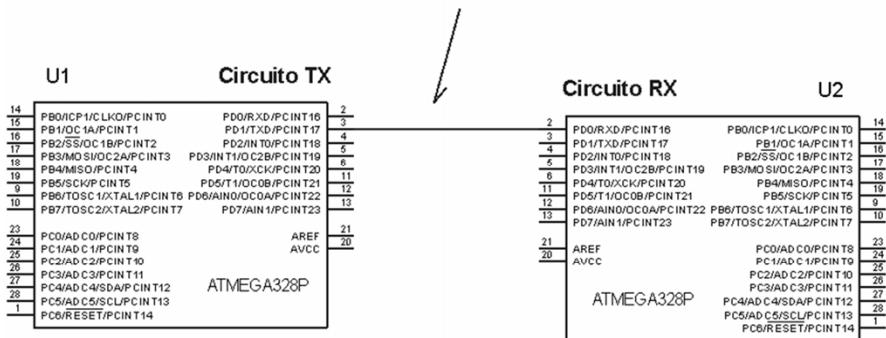
É importante utilizar antenas nos módulos de transmissão e recepção. Elas podem ser confeccionadas com um fio condutor cujo comprimento vai depender da frequência de operação, geralmente $\frac{1}{4}$ do comprimento de onda, aproximadamente 23 cm para 315 MHz e 17 cm para 433 MHz. O alcance da comunicação pode chegar a dezenas de metros, dependendo do ambiente de trabalho.

Ao se empregar microcontroladores para realizar o controle dos módulos de RF, a USART torna fácil a tarefa de gerar os dados de transmissão e recepção. Na maioria dos módulos básicos de RF, a taxa de comunicação é de poucos kbps; por exemplo, no módulo apresentado na

fig. 15.9, chega-se no máximo a 2 kbps. Esse fator é limitante em aplicações que necessitam transmitir dados a velocidades maiores.

Na fig. 15.11, é apresentado um circuito simulando a conexão entre os módulos de RF controlados por dois microcontroladores. Esse circuito é útil para elucidar o funcionamento do sistema e permitir a sua programação.

O meio físico é feito pelo link dos módulos de RF.



O pino TXD do microcontrolador U1 é ligado ao pino de dados do Módulo RF TX. O pino RXD do microcontrolador U2 é ligado ao pino de dados do Módulo RF RX.

A alimentação dos módulos de RF é de 5 V e o terra é comum ao circuito.

Fig. 15.11 – Circuito simulando a conexão entre os módulos de RF controlados por dois microcontroladores ATmega328

Para o funcionamento do circuito da fig. 15.11, foi utilizada uma taxa de comunicação de 1200 bps, comunicação assíncrona, 8 bits de dados, 1 bit de paridade par e 1 de parada. O bit de paridade é utilizado para verificar a integridade dos dados recebidos, importante devido a natureza da comunicação.

Como podem existir mais dispositivos operando na mesma faixa de frequência dos módulos de RF, é importante atribuir um endereço aos módulos. Como a comunicação é de 8 bits, podem ser empregados 4 bits para indicar o endereço e outros 4 para a transmissão do dado no mesmo

byte (se desejado pode ser empregado um endereçamento mais complexo). O programa a seguir, por exemplo, utiliza os 4 bits menos significativos para atribuir o endereço 5 aos módulos de comunicação e os 4 bits mais significativos indicam qual deve ser a operação a ser efetuada pelo circuito receptor. O programa de controle da transmissão envia constantemente 16 dados diferentes. Por sua vez, o programa de controle da recepção atribui o valor do dado recebido ao PORTB de acordo com o seu valor.

Modulo_RF_TX.c

```
//===================================================================== //
//          CONTROLE DO MÓDULO DE RF TX                         //
//===================================================================== //
#include "def_principais.h"
#include "USART.h"

#define endereco 0x05      //define o endereço de acesso para o modulo RF RX
//-----
void USART_Inic(unsigned int ubbr0);
void USART_Transmite(unsigned char dado);
//-----
int main()
{
    unsigned char i, dado;

    USART_Inic(MYUBRR); //UCSR0C = (1<<UPM01)|(1<<UCSZ01)|(1<<UCSZ00);

    while(1)           //envia continuamente
    {
        for(i=0; i<16;i++)      //envia 16 dados diferentes (são 4 MSB)
        {
            dado = (i << 4 ) | endereco; //4 LSB são o endereço do módulo RF RX

            USART_Transmite(dado); //envia 3 vezes para aumentar a chance do dado
            USART_Transmite(dado); //ser recebido
            USART_Transmite(dado);

            _delay_ms(500); //envia um novo dado a cada 0,5 s
        }
    }
} //-----
```

Modulo_RF_RX.c

```
//=====================================================================
//          CONTROLE DO MÓDULO DE RF RX
//=====================================================================
#include "def_principais.h"
#include "USART.h"

#define endereco 0x05      //define o endereço de acesso
//-----
void USART_Inic(unsigned int ubbr0);
unsigned char USART_Recebe();
//-----
int main()
{
    unsigned char dado;

    DDRB = 0xFF;//PORTB para sinalizar o dado recebido

    USART_Inic(MYUBRR);

    while(1)
    {
        dado = USART_Recebe();

        //se não houver erro de paridade executa a ação correspondente
        if(!tst_bit(UCSR0A,UPE0))
        {
            if((dado & 0x0F) == endereco) //se o dado é válido executa a ação
            {
                dado = dado>>4; //elimina o endereço do dado

                switch(dado)    //16 possibilidades (4 bits de informação)
                {
                    //cada “case” terá a ação desejada
                    case 0: PORTB = dado; break;
                    case 1: PORTB = dado; break;
                    case 2: PORTB = dado; break;
                    case 3: PORTB = dado; break;
                    case 4: PORTB = dado; break;
                    case 5: PORTB = dado; break;
                    case 6: PORTB = dado; break;
                    case 7: PORTB = dado; break;
                    case 8: PORTB = dado; break;
                    case 9: PORTB = dado; break;
                    case 10: PORTB = dado; break;
                    case 11: PORTB = dado; break;
                    case 12: PORTB = dado; break;
                    case 13: PORTB = dado; break;
                    case 14: PORTB = dado; break;
                    default: PORTB = dado;
                }
            }
        } //if de paridade
    } //while 1
}
```

Exercício:

15.4 – Faça um programa para ligar 5 LEDs em um circuito com um módulo RF RX de acordo com os botões ligados a um módulo RF TX, conforme apresentado na fig. 15.12. Isto é, faça um controle remoto onde cada botão aciona um único LED.

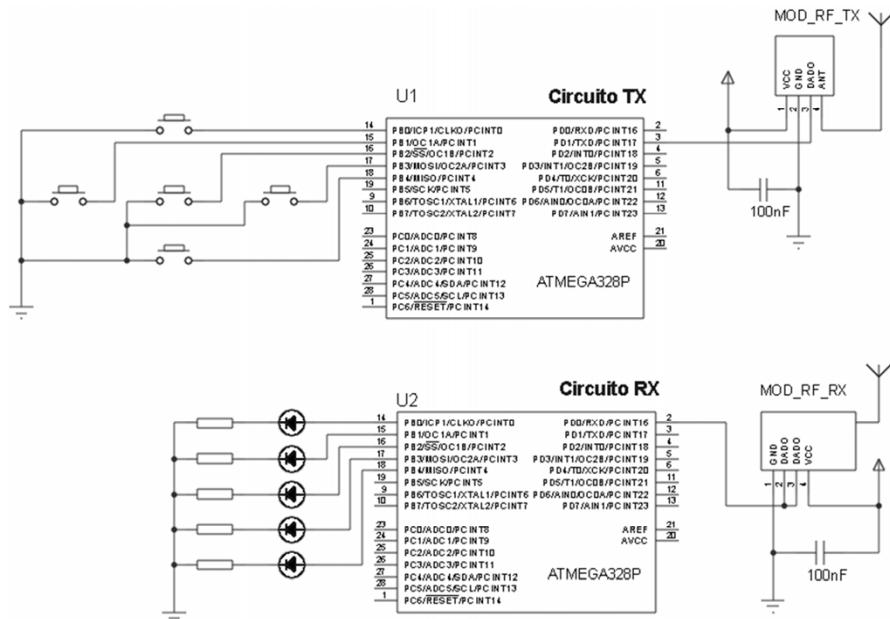


Fig. 15.12 – Controle remoto por RF.

Obs.: os capacitores de desacoplamento de 100 nF são importantes para o correto funcionamento do circuito.

15.5 MÓDULO BLUETOOTH PARA PORTA SERIAL

O bluetooth é uma tecnologia desenvolvida para ter baixo custo, apresentar baixo consumo de energia e um curto alcance de comunicação. Primariamente, visava a substituição de cabos de sinal. Atualmente, permite a troca de informações entre dispositivos, tais como: computadores, fones de ouvido, celulares, impressoras, câmeras digitais e tablets. Os dispositivos mais comuns consomem 2,5 mW e possuem um alcance de até 10 m. Entretanto, existem dispositivos que podem alcançar até 100 m com uma potência de 100 mW. Dependendo da versão do bluetooth, as taxas de comunicação podem estender-se de 1 Mbps até 24 Mbps. Ainda, o bluetooth permite implementar redes de comunicação com vários dispositivos interconectados.

O protocolo bluetooth é complexo e composto por várias camadas (pilhas de protocolos), uma parte feita por hardware e outra pelo software embarcado no dispositivo de aplicação. Opera na frequência ISM de 2,4 GHz (2,4 GHz até 2,4835 GHz) com a modulação GFSK (*Gaussian Frequency Shift Keying*), a qual emprega um algoritmo de alta velocidade de deslocamento pseudoaleatório de frequência (altera a frequência de trabalho continuamente) com uma chave de criptografia de 128 bits, o que torna a comunicação muito segura.

Para a conexão entre dois dispositivos bluetooth, o pareamento é necessário. Por exemplo, no caso de um celular e um computador, o computador irá procurar os dispositivos bluetooth disponíveis ao seu redor. O celular, por sua vez, deve estar habilitado para ser reconhecido e, então, os dispositivos trocarão uma senha para a comunicação. Com isso, o pareamento é realizado e os dispositivos podem trocar informações. Isso é feito apenas uma vez.

A forma com que o bluetooth irá trocar informações depende do tipo de protocolo empregado no envio dos dados. O interessante é que o bluetooth permite o protocolo RFCOMM (*Radio Frequency Communications*), que emula uma comunicação serial padrão (USART),

permitindo a troca de informações entre dispositivos com se fossem conectados por um cabo destinado à comunicação RS232. Em um computador, pode ser criada uma porta serial COM virtual. Então, feito o pareamento dos dispositivos, a troca de informação é idêntica a de uma USART, podendo chegar a uma taxa de comunicação de até 115,2 kbps.

Para aplicações com microcontroladores, quando se deseja utilizar o bluetooth sem a complexidade do gerenciamento de dados do seu modo padrão de trabalho, existem módulos para realizar exclusivamente o protocolo RFCOMM (bluetooth para porta serial), o que torna simples o uso do bluetooth. Na fig. 15.13, são apresentados dois desses módulos: um, montado em uma placa de circuito impresso (PCI) para ser conectada diretamente ao Arduino e outro, montado em uma PCI menor.

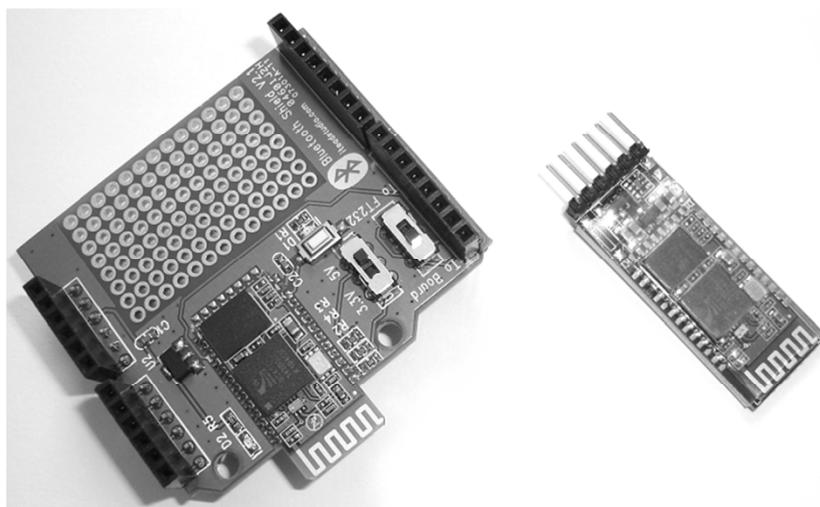


Fig. 15.13 – Bluetooth *shield* (BT Shield, iteadstudio.com) e módulo bluetooth individual.

O módulo apresentado na fig. 15.13 é facilmente reconhecido por um computador com bluetooth e o fabricante especifica como valores padrão: o código 1234 para o pareamento, taxa de comunicação de 9600 bps , 8 bits de dados, sem paridade e 1 bit de parada.

Na comunicação com um computador, após o pareamento entre os dispositivos, pode-se empregar um programa de terminal virtual (*Hyperterminal*), como apresentado na seção 15.3. A porta COM do dispositivo deve estar aberta para a comunicação.

O mesmo programa (seção 15.3) utilizado para a comunicação entre o Arduino e o computador foi usado com o Bluetooth *Shield* da fig. 15.13 e os resultados foram idênticos. O protocolo RFCOMM faz a comunicação bluetooth se tornar transparente ao usuário, como se ela fosse uma simples comunicação serial.

Com a popularização dos *Smart Phones* e *tablets* (que já possuem internamente bluetooth) são inúmeras as possibilidades de controle por sistemas microcontrolados utilizando módulos conversores serial/bluetooth.

Exercícios:

15.5 – Pesquise as aplicações realizadas com *Smart Phones* e *tablets* que utilizam bluetooth para interagir com sistemas microcontrolados. Por exemplo, um osciloscópio enviando dados via bluetooth para um celular.

15.6 – Pesquise sobre a tecnologia ZigBee. Como a USART poderia ser empregada para o trabalho com essa tecnologia?

16. TWI (TWO WIRE SERIAL INTERFACE) – I2C

Neste capítulo, é apresentada a interface de comunicação serial I2C do ATmega. Para exemplificar seu uso, é utilizado um relógio de tempo real (RTC – *Real Time Clock*), o DS1307.

Desenvolvido pela Philips nos anos 80, o protocolo I2C é um protocolo mestre/escravo utilizado para comunicação de dados em placas de circuito impresso. Todos os dispositivos são conectados através de duas vias: dados seriais (SDA) e *clock* serial (SCL). No ATmega, é denominado TWI (*Two-Wire Interface*) para o não pagamento de *royalties* pelo uso do termo I2C.

O I2C é encontrado em um grande número de circuitos integrados, em sua maioria, utilizados como circuitos periféricos em sistemas microcontrolados, tais como: memórias EEPROM, sensores de temperatura, relógios de tempo real, conversores AD e DA.

O protocolo permite vários mestres no mesmo barramento, sendo mais comum um único mestre com diversos escravos. Também é possível o endereçamento de 10 bits, o que não é usual (virtualmente poderiam ser conectados até 1024 dispositivos diferentes). O endereçamento mais empregado é o de 7 bits, permitindo a conexão de até 112 dispositivos, pois 16 endereços são reservados. O barramento I2C com N dispositivos é ilustrado na fig. 16.1.

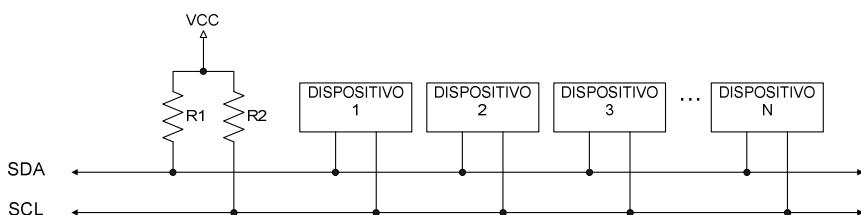


Fig. 16.1 – Barramento I2C – TWI.

Os fabricantes pagam à Philips para obter um endereço para o seu circuito integrado. Isso significa que somente um fabricante pode usar o endereço, o que é bom quando se programa, porque se pode especificar o dispositivo com o qual o mestre vai se comunicar. Por outro lado, o grave inconveniente é que não se pode colocar diretamente dois dispositivos iguais no barramento sem um hardware extra. Alguns dispositivos fornecem um meio simples de alterar o seu endereço, o que permite até 8 dispositivos iguais conectados ao barramento I2C.

O I2C utiliza duas vias para a comunicação, uma via de dados (SDA) e uma via de *clock* (SCL), ver a fig. 16.1. A via de dados transporta endereços, controle e dados, e a via de *clock* sincroniza o transmissor e o receptor durante a transferência.

Os dispositivos I2C são classificados como mestre ou escravo. O mestre gera o sinal de *clock* e inicia a transmissão; ao escravo cabe obedecer aos comandos. Um dispositivo pode ser apenas mestre, apenas escravo, ou comutar entre mestre e escravo, conforme a aplicação. Quando existem múltiplos escravos, cada um possui o seu próprio e único endereço e só responde caso selecionado.

Quando um mestre inicia uma mensagem, ele inclui o endereço do escravo no início da mensagem. Todo o dispositivo no barramento recebe a mensagem, mas somente o escravo que reconhece o seu próprio endereço participa da transferência de dados.

O I2C permite taxas de comunicação de 100 kbps (modo padrão), 400 kbps (modo rápido) e 3.4 Mbps (modo ultrarrápido). O protocolo permite uma capacidade máxima do barramento de 400 pF (sem repetidores), o que representa cerca de 20 a 30 dispositivos ou 10 metros de fio. Essa capacidade tem influência nos resistores obrigatórios utilizados como *pull-ups* nas linhas SDA e SCL (10 kΩ para uma comunicação a 100 kHz e 1 kΩ, para 400 kHz).

16.1 I2C NO ATMEGA328

Como mencionado anteriormente, o ATmega usa o termo TWI para sua interface I2C. As principais características dessa interface no ATmega328 são:

- Operação como mestre ou escravo.
- Endereçamento de 7 bits.
- Permite a operação com vários mestres.
- Velocidade de até 400 kHz na transferência de dados.
- Drivers de saída com limitação da taxa de subida (*Slew Rate*).
- Circuito supressor de ruído no barramento.
- O reconhecimento de endereço permite ‘despertar’ a CPU do modo *Sleep*.

Para evitar conflitos na via de dados (SDA) do ATmega, os pinos são chaveados como I/O dependendo das necessidades do protocolo (dreno aberto). O pino de *clock* (SCL) também é alterado como I/O. Em ambos os pinos são necessários resistores de *pull-up* (comuns ao I2C). Assim, o valor padrão para SDA e SCL é o nível lógico alto (1). Os bits são sempre lidos na borda de descida do SCL. Os dispositivos ligados ao barramento possuem endereços individuais e os mecanismos inerentes ao trabalho com o I2C gerenciam o protocolo.

Cada bit transferido para o barramento é acompanhado por um pulso da linha de *clock*. O nível lógico do dado deve estar estável quando a linha de *clock* for para o nível alto. A única exceção acontece quando se geram a condição de início e parada da comunicação. Esse fato é ilustrado na fig. 16.2.

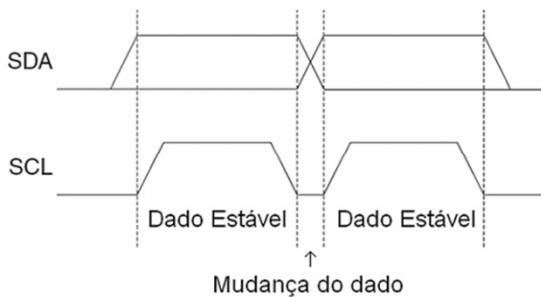


Fig. 16.2 – Validade dos dados no barramento I2C.

Os sinais possíveis no barramento são:

- **Condição de Início:** usada para informar à rede que uma transmissão irá ocorrer.
- **Condição de Parada:** usada para informar à rede que a transmissão acabou.
- **Condição de Reinício:** pode ser executada antes da parada ou na necessidade de reenvio do primeiro byte antes da finalização com a parada.
- **Condição de Acknowledge (ACK):** resposta pelo mestre ou escravo, no lugar do 9º bit, informando se o dado foi corretamente recebido ou não.
- **Transmissão de endereço:** independentemente da informação a ser transmitida (endereço de 7 bits e dados de 8 bits), a comunicação sempre é feita em pacotes de 8 bits e mais um retorno de *acknowledge*.
- **Transmissão de dados:** depois do término da transferência da parte relacionada ao endereço, o mestre ou o escravo deverão transmitir um ou mais bytes de dados.
- **Pausa:** Mantendo a linha de dados em nível baixo para processar uma informação, por exemplo.

A transmissão é iniciada quando o mestre executa uma condição de início no barramento e termina quando o mestre executa uma condição de parada. Entre o início e a parada, o barramento é considerado ocupado e nenhum outro mestre deve tentar controlar o barramento. Um caso

especial ocorre quando um novo início ocorre entre uma condição de início e parada. Isso é conhecido como reinício e é realizado quando o mestre deseja iniciar uma nova transmissão sem perder o controle sobre o barramento. Após o reinício, o barramento é considerado ocupado até a próxima parada. Esse fato é ilustrado na fig. 16.3.

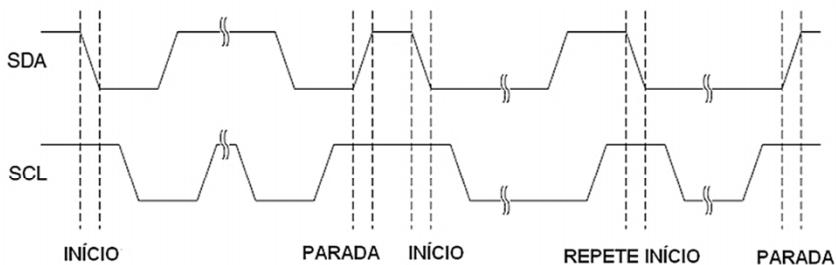


Fig. 16.3 – Condição de início, parada e reinício.

Todos os endereços transmitidos no barramento TWI são compostos por 9 bits, onde 7 são os bits referentes ao endereço, 1 bit de controle para leitura/escrita (L/E) e 1 bit de confirmação (ACK). Se o bit de leitura/escrita é 1, uma leitura será realizada, caso contrário, uma escrita. Quando o escravo reconhece que está sendo endereçado, ele sinaliza colocando a linha SDA em baixo no nono ciclo do SCL (ACK). Se o endereço do escravo estiver ocupado, ou por outra razão ele não puder atender ao mestre, a linha SDA deve ser mantida alta no ciclo de *clock* do ACK. O mestre pode, então, transmitir uma condição de parada ou de reinício para iniciar uma nova transmissão.

O bit mais significativo (MSB) do endereço ou dado é transmitido primeiro. O endereço do escravo pode ser determinado pelo projetista. Entretanto, o endereço 0x00 é reservado para uma chamada geral. Na fig. 16.4, é apresentado o formato do pacote de bits referente ao endereço.

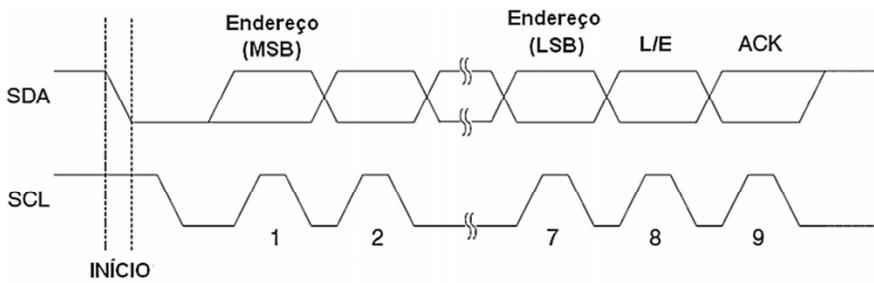


Fig. 16.4 – Formato do pacote de bits referentes ao endereço.

Quando uma chamada geral é realizada, todos os escravos devem responder colocando a linha SDA em nível baixo no ciclo de ACK. Essa chamada é realizada quando o mestre deseja transmitir a mesma mensagem aos escravos do sistema. Quando uma chamada geral é seguida por um bit de escrita, todos os escravos devem colocar a linha SDA em nível baixo no ciclo de ACK. Os pacotes seguintes de dados serão recebidos por todos os escravos que responderam. Não se deve transmitir uma chamada geral seguida de um bit de leitura, pois todos os escravos escreverão no barramento ao mesmo tempo.

Um sinal ACK é emitido ou transmitido pelo mestre ou escravo colocando a linha SDA em zero durante o nono ciclo do SCL. Se a linha for deixada em nível alto é sinalizado um NÃO-ACK (NACK). Quando o receptor receber o último byte ou por alguma razão não puder receber mais bytes, ele deve informar o transmissor enviando um NACK.

Em resumo, todos os pacotes transmitidos no barramento TWI possuem nove bits, um byte mais um bit de resposta. Durante a transmissão de dados, o mestre gera o *clock* e a condição de início e de parada, enquanto o escravo é responsável por responder se recebeu o dado.

Uma transmissão típica é apresentada na fig. 16.5. Notar que vários bytes de dados podem ser transmitidos entre o endereço do escravo mais o

bit de leitura/escrita e a condição de parada, dependendo do protocolo implementado pelo software.

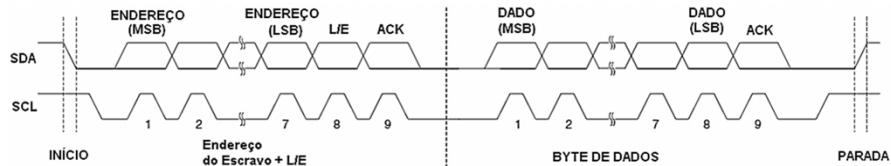


Fig. 16.5 – Transmissão de dados típica para o protocolo I2C.

16.2 REGISTRADORES DO TWI

A frequência do sinal SCL é controlada ajustando-se o registrador da taxa de bits (TWBR) e os bits de *prescaler* (divisor de *clock*) no registrador de *status* do TWI (TWSR). O modo escravo não depende desse ajuste, entretanto, a frequência da CPU deve ser no mínimo 16 vezes maior que a frequência do sinal SCL. Assim, a frequência do sinal SCL é dada por:

$$f_{\text{SCL}} = \frac{f_{\text{osc}}}{16 + (2 \times \text{TWBR} \times \text{TWPS})} \quad [\text{Hz}] \quad (16.1)$$

onde f_{osc} é a frequência de trabalho da CPU, TWBR é o valor do registrador de ajuste da taxa de bits e TWPS é o valor do *prescaler* no registrador TWSR, definido pelos bits TWPS0:1.

TWBR – TWI Bit Rate Register

Bit	7	6	5	4	3	2	1	0
TWBR	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0
Lê/Escrive	L/E							

Valor Inicial

Este registrador seleciona o fator de divisão do gerador da taxa de bits, conforme eq. 16.1.

TWSR – TWI Status Register

Bit	7	6	5	4	3	2	1	0
TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0
Lê/Escrive	L	L	L	L	L	L	L/E	L/E
Valor Inicial	1	1	1	1	1	0	0	0

Bits 7:3 – TWS7:3 – TWI Status

Estes 5 bits refletem o estado lógico do TWI. Como o registrador também contém 2 bits de *prescaler*, o software deve mascará-los durante a leitura dos bits de estado.

Bits 1:0 – TWPS1:0 – TWI Prescaler Bits

Estes bits podem ser lidos e escritos e controlam o *prescaler* da taxa de bits (eq. 16.1), conforme tab. 16.1.

Tab. 16.1 – *Prescaler* da taxa de bits do TWI.

TWPS1	TWPS0	Valor do Prescaler
0	0	1
0	1	4
1	0	16
1	1	64

TWCR – TWI Control Register

Bit	7	6	5	4	3	2	1	0
TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Lê/Escrive	L/E	L/E	L/E	L/E	L	L/E	L	L/E
Valor Inicial	0	0	0	0	0	0	0	0

É empregado para controlar a operação do TWI: habilitar, iniciar uma condição de início, gerar o ACK de recebimento, gerar uma condição de parada. Mantém o barramento em nível lógico alto enquanto o dado para o barramento é escrito no TWDR. Também indica uma colisão de dados se houver tentativa de escrita no TWDR enquanto esse estiver inacessível.

Bit 7 – TWINT – TWI Interrupt Flag

Este bit é ativo por hardware quando o TWI conclui sua atividade corrente e espera uma resposta do software de aplicação. Se o bit I do SREG e o bit TWIE do TWCR estiverem ativos, a CPU é desviada para o endereço do vetor de interrupção correspondente. Enquanto TWINT=1, o período baixo do SCL é mantido. Ele deve ser limpo por software pela escrita de 1. Limpar o bit TWINT prossegue a operação do TWI, logo os acessos aos registradores TWAR, TWSR e TWDR já devem estar completos.

Bit 6 – TWEA – TWI Enable Acknowledge Bit

Este bit controla a geração do pulso de ACK. Se for escrito 1, um pulso de ACK é gerado no barramento TWI se as seguintes condições forem encontradas:

- O endereço próprio do dispositivo como escravo tenha sido recebido.

2. Uma chamada geral tenha sido recebida, enquanto o bit TWGCE do TWAR estiver em 1.

3. Um byte de dados tenha sido recebido no modo mestre ou escravo.

Zerando o bit, o microcontrolador pode ser virtualmente desconectado por um intervalo de tempo do barramento TWI. O reconhecimento de endereço pode ser reabilitado escrevendo-se 1 no bit.

Bit 5 – TWSTA – TWI Start Condition Bit

Este bit ativo indica que o modo de comunicação será mestre. Deve ser limpo por software após uma condição de início ser transmitida.

Bit 4 – TWSTO – TWI Stop Condition Bit

Quando em 1, gera uma condição de parada, sendo limpo automaticamente pelo hardware. No modo escravo, pode ser utilizado para a recuperação de uma condição de erro, não gerando uma condição de parada, mas colocando os pinos do barramento em alta impedância e em uma condição de não endereçamento.

Bit 3 – TWWC – TWI Write Collision Flag

Este bit é ativo quando se tenta escrever no registrador de dados TWDR enquanto TWINT estiver em 0. É limpo escrevendo-se em TWDR quando TWINT estiver em 1.

Bit 2 – TWEN – TWI Enable Bit

Este bit habilita a operação do TWI e ativa a interface TWI.

Bit 0 – TWIE – TWI Interrupt Enable

Quando este bit estiver em 1, bem como o bit I do registrador SREG, a interrupção do TWI estará ativa pelo tempo que o bit TWINT estiver em nível lógico alto.

TWDR – TWI Data Register

Bit	7	6	5	4	3	2	1	0
TWDR	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0
Lê/Escrive	L/E							
Valor Inicial	1	1	1	1	1	1	1	1

No modo de transmissão, o TWDR contém o próximo byte a ser transmitido. No modo de recepção, o TWDR contém o último byte recebido. Ele pode ser escrito se o TWI não estiver enviando ou recebendo um byte. O bit ACK é controlado automaticamente pelo TWI e a CPU não pode acessá-lo diretamente.

TWAR – TWI (Slave) Address Register

Bit	7	6	5	4	3	2	1	0
TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
Lê/Escrive	L/E							
Valor Inicial	1	1	1	1	1	1	1	0

Este registrador deve ser carregado com o endereço de 7 bits que identificará o ATmega328 no barramento TWI quando no modo escravo.

O bit 0 (TWGCE) é usado para habilitar o reconhecimento de chamada geral (0x00). Existe um comparador de endereço que gera um pedido de interrupção se houver igualdade no endereço do escravo ou se a chamada geral estiver habilitada.

16.3 USANDO O TWI

O TWI é orientado a byte e baseado em interrupções. Interrupções são feitas após a ocorrência de eventos no barramento, como a recepção de um byte ou a transmissão de uma condição de inicio. Sendo o TWI baseado em interrupções, o software fica livre para executar outras operações durante a transferência de bytes. Um exemplo simples da aplicação do TWI é apresentado na fig. 16.6, no qual o mestre deseja transmitir um simples byte de dados para o escravo.

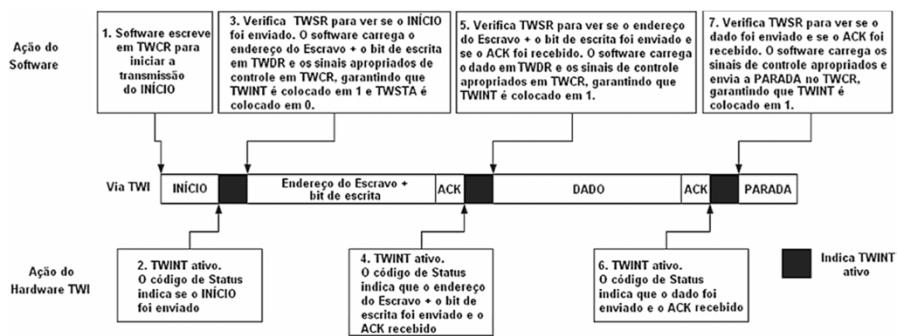


Fig. 16.6 – Transmissão típica com o TWI.

O código a seguir, sem o uso de interrupção, ilustra a programação para o exemplo acima, conforme manual do fabricante.

```

//=====
//          CÓDIGO EXEMPLO PARA USO DO TWI
//=====

TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN); //Envia a condição de início
//-----
//      while (!(TWCR & (1<<TWINT))); //Espera o TWINT ser ativo indicando que a
//      //condição de início foi transmitida
//-----
if ((TWSR & 0xF8) != START) //Verifica o valor do TWI no registrador de status.
    ERROR(); /*Mascara os bits do prescaler. Se o status
    for diferente da condição de início chama
    uma função para o tratamento do erro*/
//-----
TWDR = SLA_W; //Carrega o endereço do escravo para a escrita
TWCR = (1<<TWINT) | (1<<TWEN); /*limpa o bit TWINT no TWCR para começar a
transmissão do endereço*/
//-----
while (!(TWCR & (1<<TWINT))); /*Espera pela ativação do bit TWINT
indicando que o endereço do escravo + o bit de
escrita foi enviado e que o ACK/NACK foi recebido*/
//-----
if ((TWSR & 0xF8) != MT_SLA_ACK)//Verifica o valor do registrador de
    ERROR(); /*status do TWI. Mascara os bits do prescaler. Se o
    status for diferente de MT_SLA_ACK chama uma
    função para o tratamento do erro*/
//-----
TWDR = DATA; //Carrega DATA no registrador TWDR. Limpa o bit TWINT no TWCR
TWCR = (1<<TWINT) | (1<<TWEN); //para iniciar a transmissão do dado.
//-----
while (!(TWCR & (1<<TWINT)));/*Espera o bit TWINT ser ativo, indicando que
o dado foi transmitido e que o ACK/NACK foi recebido*/
//-----
if ((TWSR & 0xF8) != MT_DATA_ACK) //Verifica o valor do registrador de
    ERROR(); /*status do TWI. Mascara os bits do prescaler. Se o
    status for diferente de MT_DATA_ACK chama uma função
    para o tratamento do erro*/
//-----
TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWSTO); //Transmite a condição de parada
//=====

```

Quando se utiliza o AVR-GCC como compilador, pode-se empregar sua biblioteca para o trabalho com o TWI. Nela encontram-se as definições dos estados que podem ocorrer no registrador TWSR. Toda e qualquer operação que é realizada pelo módulo TWI será armazenada nesse registrador de *status*. Basta, então, analisar o seu conteúdo para se saber o que aconteceu. O TWSR indicará sucesso ou erro na transmissão/recepção dos dados (ver o manual do ATmega para maiores detalhes). A seguir, é apresentado um conjunto de definições que podem facilitar a vida do programador para o emprego do módulo TWI. As definições de teste verificam que houve erro na ação efetuada e chamam uma rotina para o tratamento do erro. Essa rotina deve ser escrita pelo programador e dependerá de como ele deseja tratar o erro, como, por exemplo, repetir um comando mal sucedido.

```

#include <util/twi.h> //definições para o uso da interface i2c
...
//-----
// Definições para o uso da comunicação I2C
//-----
#define start_bit() TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN)
#define stop_bit() TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO)
#define espera_envio() while (!(TWCR & (1<<TWINT)))
#define envia_byte() TWCR = (1<<TWINT) | (1<<TWEN)
#define recebe_byte() TWCR = (1<<TWINT) | (1<<TWEN)
#define espera_recebimento() while (!(TWCR & (1<<TWINT)))
#define recebe_byte_ret_nack() TWCR = (1<<TWINT) | (1<<TWEN)

//A rotina de tratamento de erro é por conta do programador e da sua lógica de programação

#define teste_envio_start() if((TWSR & 0xF8) != TW_START) trata_erro();
#define teste_envio_end_escrita() if((TWSR & 0xF8) != TW_MT_SLA_ACK) trata_erro();
#define teste_envio_dado() if((TWSR & 0xF8) != TW_MT_DATA_ACK) trata_erro();
#define teste_envio_restart() if((TWSR & 0xF8) != TW REP START) trata_erro();
#define teste_envio_end_leitura() if((TWSR & 0xF8) != TW_MR_SLA_ACK) trata_erro();
#define teste_recebe_byte_ret_nack() if((TWSR & 0xF8) != TW_MR_DATA_NACK) trata_erro();
//-----

```

O TWI pode operar em 4 modos principais: Mestre Transmissor (MT), Mestre Receptor (MR), Escravo Transmissor (ST) e Escravo Receptor (SR). Vários desses modos podem ser empregados em uma mesma aplicação. Por exemplo, o modo MT pode ser usado para escrever dados em um EEPROM I2C e o modo MR para ler dados da mesma. Existe a possibilidade da existência de múltiplos mestres no barramento. Caso ocorra o início simultâneo de uma transmissão por dois ou mais deles, o TWI deve garantir que um deles efetue a transmissão, sem ocasionar a perda de dados.

Os 4 modos serão descritos sucintamente de acordo com as tabelas e figuras a seguir. Foram consideradas as abreviações conforme o manual do ATmega328:

S = condição de início (*Start condition*)

Rs = repetição da condição de início (*Repeated start condition*)

R = bit de leitura (nível lógico alto no pino SDA) – *Read bit*

W = bit de escrita (nível lógico baixo no pino SDA) – *Write bit*

- A** = bit de reconhecimento (nível lógico baixo no pino SDA) – *Acknowledge bit*
- Ā** = bit de reconhecimento negado (nível lógico alto no pino SDA) – *Not Acknowledge bit*

Data= byte com 8 bits de **Dados**

P = condição de parada (*stop condition*)

SLA = endereço do escravo (**SLave Address**)

Tab. 16.2 – Códigos de *status* para o modo mestre transmissor.

Código de Status (TWSR). Bits de prescaler são 0.	Status da via e da interface de Hardware TWI	Resposta do Software de Controle				Próxima ação do módulo TWI	
		Para/do TWDR	Para o TWCR				
			STA	STO	TWIN	TWE	
0x08	Uma condição de início foi transmitida.	Carga do SLA+W	0	0	1	X	SLA+W será transmitido e um ACK ou NÃO ACK recebido.
0x10	Uma condição de repetição de início foi transmitida.	Carga do SLA+W ou SLA+R	0	0	1	X	SLA+W será transmitido e um ACK ou NÃO ACK recebido.
			0	0	1	X	SLA+R será transmitido. A lógica irá mudar para o modo de recepção mestre.
0x18	SLA+W foi transmitido e o ACK foi recebido	Carga do byte de dados ou nenhuma ação TWDR	0	0	1	X	O byte de dados será transmitido e um ACK ou NÃO ACK recebido.
			1	0	1	X	Repetição da condição de início.
			0	1	1	X	A parada será transmitida e o bit TWSTO será limpo.
			1	1	1	X	Uma parada seguida de um início serão transmitidos e o bit TWSTO será limpo.
0x20	SLA+W foi transmitido e um NÃO ACK recebido	Carga do byte de dados ou nenhuma ação TWDR	0	0	1	X	O byte de dados será transmitido e um ACK ou NÃO ACK recebido.
			1	0	1	X	Repetição da condição de início.
			0	1	1	X	A parada será transmitida e o bit TWSTO será limpo.
			1	1	1	X	Uma parada seguida de um início serão transmitidos e o bit TWSTO será limpo.
0x28	O byte de dados foi transmitido e o ACK recebido	Carga do byte de dados ou nenhuma ação TWDR	0	0	1	X	O byte de dados será transmitido e um ACK ou NÃO ACK recebido.
			1	0	1	X	Repetição da condição de início.
			0	1	1	X	A parada será transmitida e o bit TWSTO será limpo.
			1	1	1	X	Uma parada seguida de um início serão transmitidos e o bit TWSTO será limpo.
0x30	O byte de dados foi transmitido e o NÃO ACK recebido	Carga do byte de dados ou nenhuma ação TWDR	0	0	1	X	O byte de dados será transmitido e um ACK ou NÃO ACK recebido.
			1	0	1	X	Repetição da condição de início.
			0	1	1	X	A parada será transmitida e o bit TWSTO será limpo.
			1	1	1	X	Uma parada seguida de um início serão transmitidos e o bit TWSTO será limpo.
0x38	Perda de controle no SLA+W ou no byte de dados	nenhuma ação TWDR	0	0	1	X	A via TWI será liberada e o modo de endereçamento escravo não será ativo.
			1	0	1	X	Uma condição de início sera enviada quando a via ficar livre.

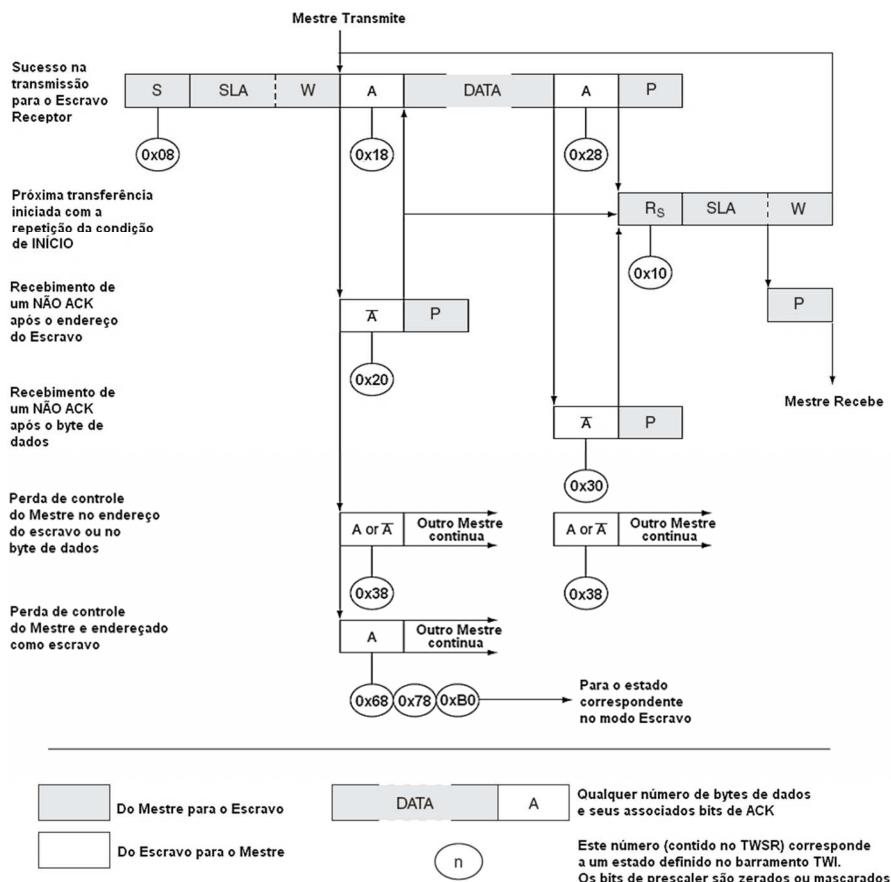


Fig. 16.7 – Formatos e estados do modo mestre transmissor.

Tab. 16.3 – Códigos de *status* para o modo mestre receptor.

Código de Status (TWSR). Bits de prescaler são 0.	Status da via e da interface de Hardware TWI	Resposta do Software de Controle				Próxima ação do módulo TWI	
		Para/do TWDR	Para o TWCR				
			STA	STO	TWINT	TWEA	
0x08	Uma condição de início foi transmitida.	carga do SLA+R	0	0	1	X	SLA+R será transmitido e um ACK ou NÃO ACK recebido.
0x10	Uma condição de repetição de início foi transmitida.	carga do SLA+R ou SLA+W	0	0	1	X	SLA+R será transmitido e um ACK ou NÃO ACK recebido.
			0	0	1	X	SLA+W será transmitido. A lógica irá mudar para o modo de transmissão mestre.
0x38	Perda de controle no SLA+R ou no bit NÃO ACK	nenhuma ação TWDR	0	0	1	X	A via TWI será liberada e o modo de endereçamento escravo não será ativo.
			1	0	1	X	Uma condição de início será enviada quando a via ficar livre.
0x40	SLA+R foi transmitido e um ACK recebido	nenhuma ação TWDR	0	0	1	0	O byte de dados será recebido e um NÃO ACK retornado.
			0	0	1	1	O byte de dados será recebido e um ACK retornado.
0x48	SLA+R foi transmitido e um NÃO ACK recebido	nenhuma ação TWDR	1	0	1	X	Repetição da condição de início.
			0	1	1	X	A parada será transmitida e o bit TWSTO será limpo.
0x50	O byte de dados foi recebido e um ACK retornado.	Leitura do byte de dados	0	0	1	0	Uma parada seguida de um início serão transmitidos e o bit TWSTO será limpo.
			0	0	1	1	O byte de dados será recebido e um NÃO ACK retornado.
0x58	O byte de dados foi recebido e um NÃO ACK retornado.	Leitura do byte de dados	1	0	1	X	Repetição da condição de início.
			0	1	1	X	A parada será transmitida e o bit TWSTO será limpo.
			1	1	1	X	Uma parada seguida de um início serão transmitidos e o bit TWSTO será limpo.

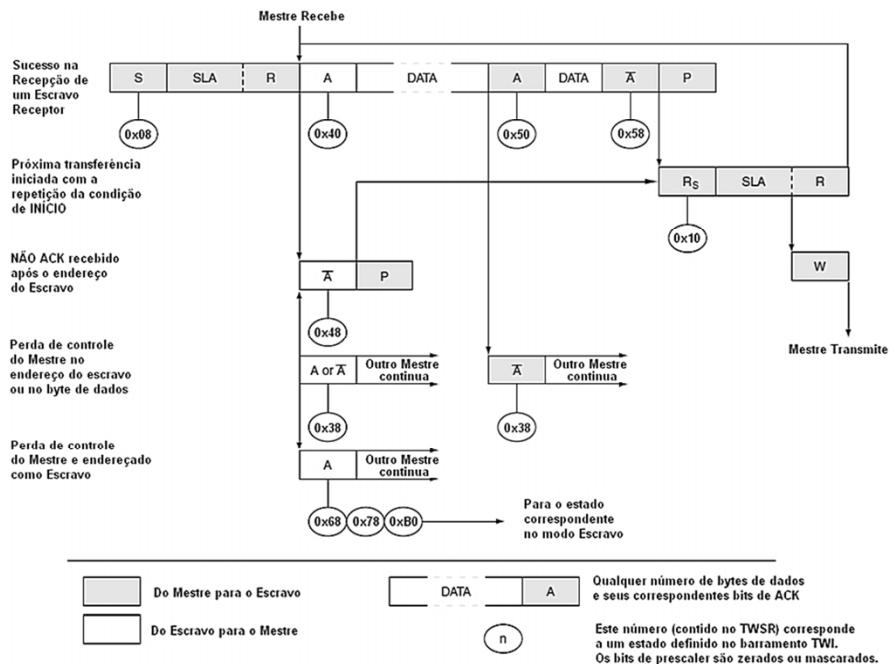


Fig. 16.8 – Formatos e estados do modo mestre receptor.

Tab. 16.4 – Códigos de *status* para o modo escravo receptor.

Código de Status (TWSR). Bits de prescaler são 0.	Status da via e da interface de Hardware TWI	Resposta do Software de Controle				Próxima ação do módulo TWI	
		Para/do TWDR	Para o TWCR				
			STA	STO	TWINT		
0x60	O próprio endereço SLA+W foi recebido e um ACK retornado.	nenhuma ação TWDR	X	0	1	0	O byte de dados será recebido e um NÃO ACK retornado. O byte de dados será recebido e um ACK retornado.
0x68	Perda de controle como mestre no SLA+R/W. O próprio SLA+W foi recebido e um ACK retornado.	nenhuma ação TWDR	X	0	1	0	O byte de dados será recebido e um NÃO ACK retornado. O byte de dados será recebido e um ACK retornado.
0x70	Uma chamada geral de endereço foi recebida e um ACK retornado.	nenhuma ação TWDR	X	0	1	0	O byte de dados será recebido e um NÃO ACK retornado. O byte de dados será recebido e um ACK retornado.
0x78	Perda de controle como mestre no SLA+R/W. A chamada geral de endereço foi recebida e um ACK retornado.	nenhuma ação TWDR	X	0	1	0	O byte de dados será recebido e um NÃO ACK retornado. O byte de dados será recebido e um ACK retornado.

0x80	Endereçado previamente com o próprio SLA+W. Um dado foi recebido e um ACK retornado.	Leitura do byte de dados	X X	0 0	1 1	0 1	O byte de dados será recebido e um NÃO ACK retornado. O byte de dados será recebido e um ACK retornado.
0x88	Endereçado previamente com o próprio SLA+W. Um dado foi recebido e um NÃO ACK retornado.	Leitura do byte de dados	0	0	1	0	Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA (<i>General Call Address</i>). Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1.
			0	0	1	1	Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Uma condição de início será transmitida quando a via ficar livre.
			1	0	1	0	Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Uma condição de início será transmitida quando a via ficar livre.
			1	0	1	1	Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1. Uma condição de inicio será transmitida quando a via ficar livre.
0x90	Endereçado previamente pela chamada geral. Um dado foi recebido e um ACK retornado.	Leitura do byte de dados	X X	0 0	1 1	0 1	O byte de dados será recebido e um NÃO ACK retornado. O byte de dados será recebido e um ACK retornado.
0x98	Endereçado previamente pela chamada geral. Um dado foi recebido e um NÃO ACK retornado.	Leitura do byte de dados	0	0	1	0	Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1.
			0	0	1	1	Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Uma condição de início será transmitida quando a via ficar livre.
			1	0	1	0	Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Uma condição de inicio será transmitida quando a via ficar livre.
			1	0	1	1	Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1. Uma condição de inicio será transmitida quando a via ficar livre.
0xA0	Uma parada ou repetição de início foi recebida enquanto ainda endereçado como escravo.	Nenhuma ação	0 0 1 1	0 0 1 0	1 1 1 1	0 1 0 1	Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1. Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Uma condição de início será transmitida quando a via ficar livre. Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1. Uma condição de inicio será transmitida quando a via ficar livre.

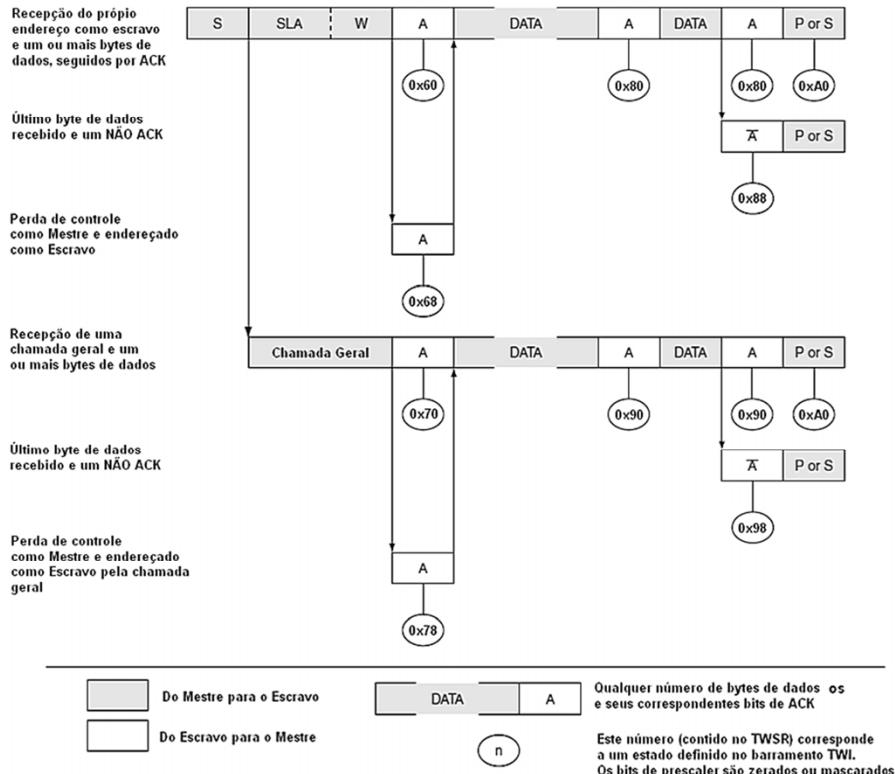


Fig. 16.9 – Formatos e estados do modo escravo receptor.

Tab. 16.5 – Códigos de *status* para o modo escravo transmissor.

Código de Status (TWSR). Bits de prescaler são 0.	Status da via e da interface de Hardware TWI	Resposta do Software de Controle				Próxima ação do módulo TWI	
		Para/do TWDR	Para o TWCR				
			STA	STO	TWINT	TWEA	
0xA8	O próprio endereço SLA+R foi recebido e um ACK retornado.	carrega o byte de dados	X X	0 0	1 1	0	O último byte de dados será transmitido e um NÃO ACK deve ser recebido. O byte de dados será transmitido e um ACK deve ser recebido.
0xB0	Perda de controle como mestre no SLA+RW. O próprio SLA+R foi recebido e um ACK retornado.	carrega o byte de dados	X X	0 0	1 1	0	O último byte de dados será transmitido e um NÃO ACK deve ser recebido. O byte de dados será transmitido e um ACK deve ser recebido.
0xB8	O byte de dados do TWDR foi transmitido e um ACK recebido	carrega o byte de dados	X X	0 0	1 1	0	O último byte de dados será transmitido e um NÃO ACK deve ser recebido. O byte de dados será transmitido e um ACK deve ser recebido.
0xC0	O byte de dados do TWDR foi transmitido e um ACK recebido	nenhuma ação TWDR	0 0 1 1	0 0 1 0	1 1	0	Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1. Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Uma condição de início será transmitida quando a via ficar livre. Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1. Uma condição de inicio será transmitida quando a via ficar livre.
0xC8	Último byte de dados do TWDR foi transmitido (TWEA=0) e um ACK foi recebido.	nenhuma ação TWDR	0 0 1 1	0 0 1 0	1 1	0	Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1. Mudança para o modo não endereçado escravo. Não reconhecimento do próprio SLA ou GCA. Uma condição de inicio será transmitida quando a via ficar livre. Mudança para o modo não endereçado escravo. O próprio SLA será reconhecido. GCA será reconhecido se TWGCE = 1. Uma condição de inicio será transmitida quando a via ficar livre.

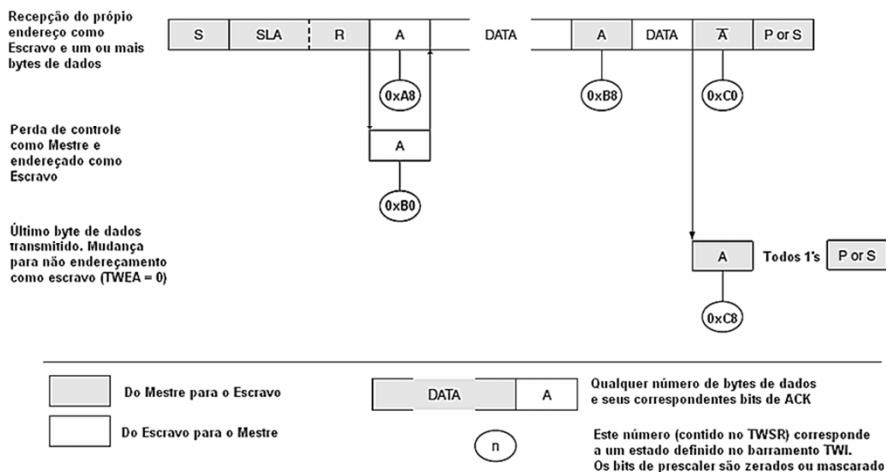


Fig. 16.10 – Formatos e estados do modo escravo transmissor.

16.4 RELÓGIO DE TEMPO REAL DS1307

O circuito integrado DS1307 é um relógio de tempo real de baixo consumo com calendário completo até o ano de 2100, que emprega o protocolo I₂C. Para sua operação, é necessário o emprego de um cristal externo de 32,768 kHz e uma bateria que garante sua operação na falta da tensão de alimentação. O CI fará a divisão adequada (por 2¹⁵) para obter a base de tempo de 1 s. Na fig. 16.11, é apresentado um módulo comercial com o DS1307, pronto para o uso.

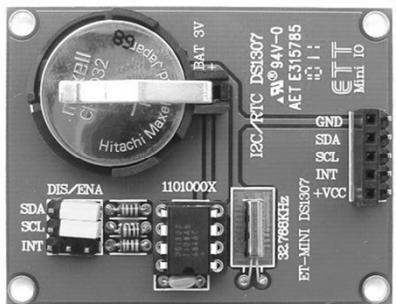


Fig. 16.11 – Módulo com o DS1307 para uso com microcontroladores (www.futurlec.com).

O mapa da memória do DS1307 é apresentado na fig. 16.12. Os dados na sua maioria empregam a codificação BCD, permitindo a leitura direta dos dois dígitos do tempo (4 LSB e 4 MSB). No endereço 0x07, é possível habilitar uma onda quadrada no pino SOUT (dreno aberto) com frequência de 1 Hz, 4096 Hz, 8192 Hz ou 32768 Hz. O bit SQWE habilita esse sinal e os bits RS0:RS1 a sua frequência de saída.

	Bit	7	6	5	4	3	2	1	0	
SEGUNDOS	0x00	CH	10.segmentos		segundos					
MINUTOS	0x01	0	10.minutes		minutes					
HORAS	0x02	0	12	24	10.h	10.h	horas			
DIA	0x03	0	0	0	0	0	dia			
DATA	0x04	0	0	10.data		data				
MÊS	0x05	0	0	0	10.	mês	mês			
ANO	0x06	10.ano			ano					
CONTROLES	0x07	OUT	0	0	SQWE	0	0	RS1	RS0	
RAM 56 x 8	0x08									
	0x3F									

Fig. 16.12 – Mapa de memória do DS1307.

Na fig. 16.13, é apresentado o formato do protocolo I2C para escrita e leitura do DS1307 conforme o manual do fabricante. Após o bit de inicio, o endereço do componente vem acompanhado de um bit indicando uma operação de escrita ou leitura.

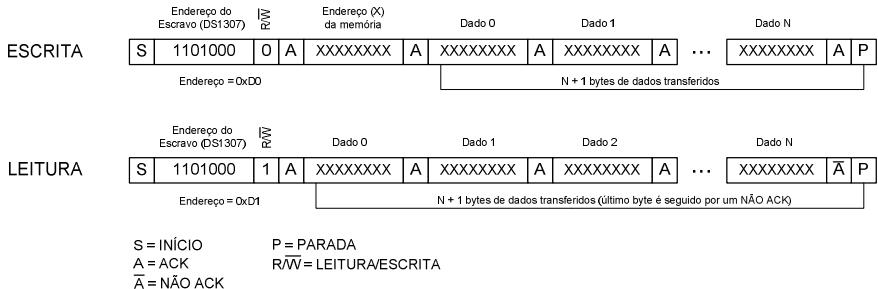


Fig. 16.13 - Formato do protocolo de comunicação I2C para o DS1307.

Um detalhe importante é que, em uma operação de leitura, é necessário primeiro realizar uma operação de escrita para atualizar o endereço de início da leitura. Na fig. 16.14, é apresentado o formato do protocolo para a leitura e escrita de um único byte de dados no DS1307.

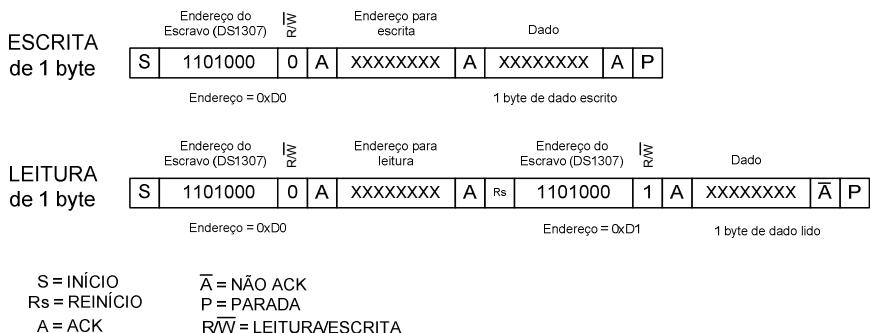


Fig. 16.14 - Formato do protocolo de comunicação I2C para a escrita e leitura de um único byte de dados no DS1307.

Abaixo, são apresentadas sub-rotinas exemplo para a escrita e leitura de 1 byte no DS1307 (suas definições foram apresentadas anteriormente). Para o emprego das sub-rotinas com outros CIs, basta trocar o endereço para a comunicação. As sub-rotinas enviam e recebem apenas 1 byte de informação. O protocolo I2C, entretanto, permite o envio e recepção de vários bytes, sem a necessidade de novos *start* bits e do endereço do componente entre os bytes.

```
-----  
//Sub-rotina para ler apenas um byte do barramento I2C do DS1307  
-----  
unsigned char le_RTC(unsigned char endereco)  
{  
    start_bit();  
    espera_envio();  
    teste_envio_start();  
  
    TWDR = 0xD0;    //carrega o endereço para acesso do DS1307 (bit0 = 0, escrita)  
                  //para outro CI basta trocar este endereço  
    envia_byte();  
    espera_envio();  
    teste_envio_end_escrita();  
  
    TWDR = endereco;    //ajuste do ponteiro de endereço para a leitura do DS1307  
  
    envia_byte();  
    espera_envio();  
    teste_envio_dado();  
  
    start_bit();    //reinício  
    espera_envio();  
    teste_envio_restart();  
  
    TWDR = 0xD1;    //carrega o endereço para acesso do DS1307 (bit0 = 1 é leitura)  
                  //automaticamente o ATmega chaveia para o estado de recepção  
    envia_byte();  
    espera_envio();  
    teste_envio_end_leitura();  
  
    recebe_byte_ret_nack();    //só lê um byte, por isso retorna um NACK  
    espera_recebimento();  
    teste_recebe_byte_ret_nack();  
  
    stop_bit();  
    return TWDR;    //retorna byte recebido  
}  
-----
```

```

//-----
//Sub-rotina para escrever apenas um byte no barramento I2C do DS1307
//-----
void escreve_RTC(unsigned char dado_i2c, unsigned char endereco)
{
    start_bit();
    espera_envio();
    teste_envio_start();

    TWDR = 0xD0;      //carrega o endereço para acesso do DS1307 (bit0 = 0 é escrita)
                      //para outro CI basta trocar este endereço
    envia_byte();
    espera_envio();
    teste_envio_end_escrita();

    TWDR = endereco;   //carrega o endereço para escrita do dado no DS1307

    envia_byte();
    espera_envio();
    teste_envio_dado();

    TWDR = dado_i2c;   //carrega o dado para escrita no endereço especificado

    envia_byte();
    espera_envio();
    teste_envio_dado();

    stop_bit();
}
//-----

```

As sub-rotinas anteriores seguem um fluxo sequencial, ou seja, o programa executa um passo por vez na sequência de escrita, o que não permite que o microcontrolador execute outra ação durante esse trabalho. Uma abordagem mais eficiente seria utilizar dentro da interrupção do TWI um programa funcionando como uma máquina de estados. Dessa forma, as ações do TWI não precisam seguir um fluxo sequencial e permitem que o programa principal possa executar outras atividades em paralelo. Na fig. 16.15, é apresentado o algoritmo no formato de máquina de estados para a escrita e leitura de um único byte no DS1307. O algoritmo prevê a correção de erros durante a execução do protocolo I2C com a repetição de um comando mal sucedido. Todo o algoritmo é feito dentro da sub-rotina de interrupção do módulo TWI.

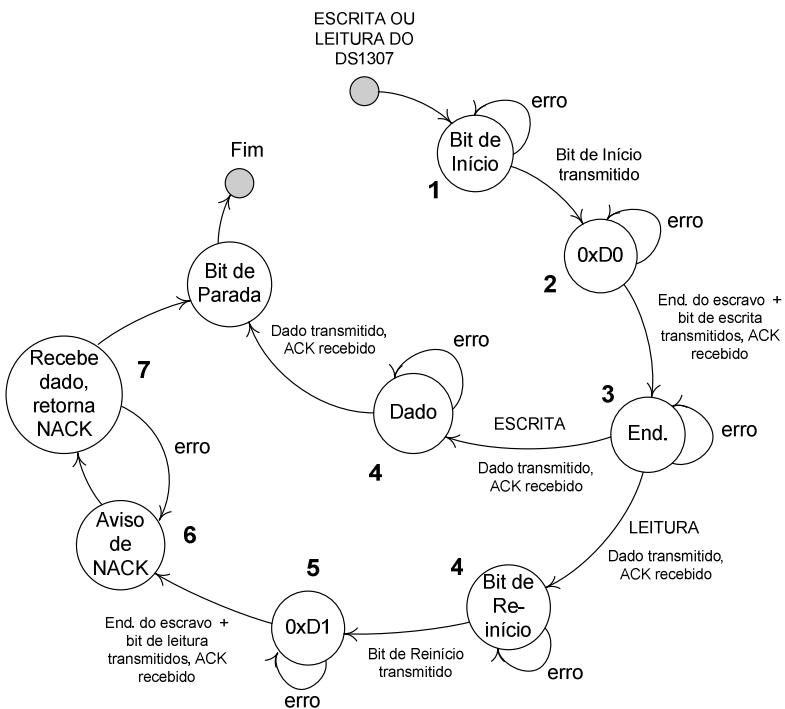


Fig. 16.15 - Máquina de estado representando o algoritmo para a escrita e leitura de um único byte no DS1307.

Além da escrita e leitura do DS1307, o microcontrolador deve ser capaz de ajustar e apresentar o horário. Na fig. 16.16, é apresentado um circuito para realizar essa tarefa. No circuito, empregam-se dois botões para o ajuste do tempo e um LCD 16 × 2. Na sequência, é apresentado o programa de controle do sistema, testado com o Arduino. É possível acompanhar o algoritmo da sub-rotina de interrupção do TWI na fig. 16.15. Os erros são tratados em conjunto no final do código, mas a lógica não se altera. Foi utilizado um contador (decrescente) que permite no máximo 256 tentativas de correções de erro, impedindo o travamento das atividades do TWI em algum estágio da correção de um possível erro.

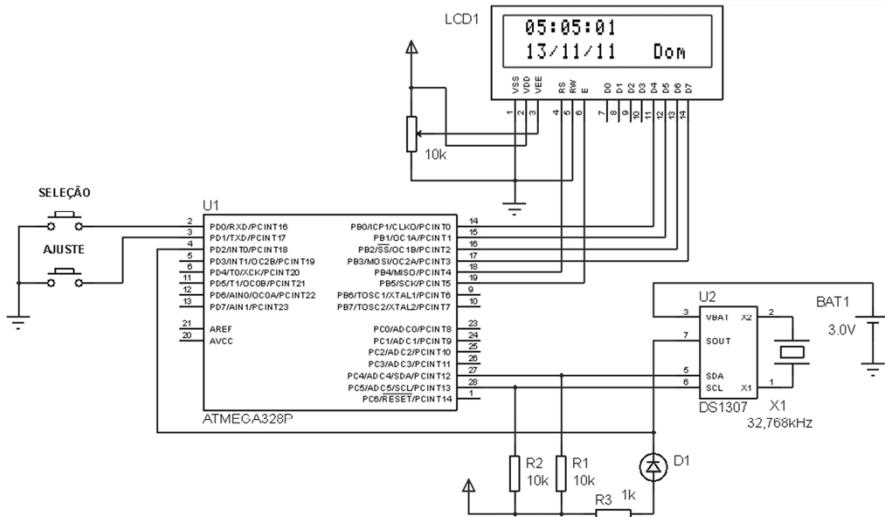


Fig. 16.16 - Circuito para o uso do DS1307.

def.principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL //define a frequência do microcontrolador - 16MHz

#include <avr/io.h> //definições do componente especificado
#include <util/delay.h> //biblioteca para o uso das rotinas de delay
#include <avr/pgmspace.h> //para o uso do PROGMEM, gravação de dados na memória flash
#include <avr/interrupt.h> //para uso das interrupções
#include <util/twi.h> //para uso das definições do TWI

//Definições de macros para o trabalho com bits
#define set_bit(y,bit) (y|=1<<bit)//coloca em 1 o bit x da variável Y
#define clr_bit(y,bit) (y&=~(1<<bit))//coloca em 0 o bit x da variável Y
#define cpl_bit(y,bit) (y^=(1<<bit))//troca o estado lógico do bit x da variável Y
#define tst_bit(y,bit) (y&(1<<bit))//retorna 0 ou 1 conforme leitura do bit

//botões para ajuste do tempo
#define SELECAO PD0
#define AJUSTE PD1

#endif
```

RTC_DS1307.c (programa principal)

```
----- //  
//      DS1307 - RTC com LCD 16 x 2 utilizando I2C          //  
//----- //  
#include "def_principais.h"  
#include "LCD.h"  
#include "DS1307.h"  
  
extern unsigned char flag_pontos, cont;  
  
int main()  
{  
    DDRB = 0xFF; //LCD  
    DDRD = 0x00; //botões  
    PORTD = 0x03;//pull-ups habilitados  
  
    //TWI  
    DDRC = (1<<4)|(1<<5);  
  
    inic_TWI();  
    sei();  
  
    inic_LCD_4bits();  
  
    escreve_DS1307(0x00, 0x00);  
    escreve_DS1307(0x07, 0b00010000); //habilita 1Hz no pino SOUT do DS1307  
  
    //-----  
    while(1)  
    {  
        ler_convert_tudo(0x00); //lê toda a memória de tempo do RTC  
        mostra_tempo();  
        mostra_pontos(flag_pontos); //flag indica se deve ligar os pontos e traços  
        _delay_ms(200);  
  
        if(!tst_bit(PIND,SELECAO))  
        {  
            flag_pontos = 0; //avisa para não imprimir mais os pontos e traços  
            cont++; //conta o nr. de vezes que o botão SELECAO foi pressionado  
  
            if (cont>7)  
            {  
                cont = 0;  
                flag_pontos = 1; //habilita novamente os pontos e traços  
            }  
            alerta_display(cont); //coloca a seta no local para ajuste do tempo  
            while(!tst_bit(PIND,SELECAO)); //aguarda soltar SELECAO  
        }  
        //-----  
        if (flag_pontos==0)  
        {  
            if(!tst_bit(PIND,AJUSTE))  
                ajusta_tempo2(cont); //ajusta RTC ou ajusta alarme  
            } //if !flag  
        //-----  
    } //while forever  
    //-----  
}
```

DS1307.h (arquivo de cabeçalho do DS1307.c)

```
#ifndef _DS1307_H
#define _DS1307_H

#include "def_principais.h"

#define start_bit()      TWCR |= (1<<TWSTA)
#define stop_bit()       TWCR |= (1<<TWSTO)
#define clr_start_bit()  TWCR &= ~(1<<TWSTA)

//TWI
void inic_TWI();
void escreve_DS1307(unsigned char end_escrita, unsigned char dado);
unsigned char le_DS1307(unsigned char end_leitura);
ISR(TWI_vect);

//trabalho com o LCD para mostrar o tempo
void converte2BCD(unsigned char dado, unsigned char endereco);
void ler_convert_tudo(unsigned char ende);
void ajusta_tempo(unsigned char end, unsigned char limite,unsigned char ini_nr1);
void ajusta_tempo2(unsigned char qual);
void mostra_tempo();
void mostraPontos(unsigned char desliga);
void alerta_display(unsigned char qual);

#endif
```

DS1307.c (aqui estão todas as funções para o trabalho⁴⁰ com o DS1307, incluindo o LCD, que é a maior parte do código)

```
#include "DS1307.h"
#include "LCD.h"

volatile unsigned char escrita, oper_TWI_concl, end_escr_leit, dado_escr, dado_leit,
                           passo, cont_max_erro;
unsigned char tempo[7], segs[14], aux_disp[4], flagPontos = 1, cont = 0;
char escr[4];
//-----
void inic_TWI() //SCL = 100 kHz (limite do DS1307) com F_CPU = 16 MHz
{
    //Ajuste da frequência de trabalho - SCL = F_CPU/(16+2.TWBR.Prescaler)
    TWBR = 18;
    TWSR |= 0x01;//prescaler = 4;
    TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWIE); //habilita o TWI e a interrupção
}
//-----
void escreve_DS1307(unsigned char end_escrita, unsigned char dado)
{
    //passa variáveis da função para as variáveis globais alteradas na ISR
    escrita = 1;           //1 para escrita, 0 para leitura
    end_escr_leit = end_escrita;
    dado_escr = dado;
```

⁴⁰ A parte de tratamento de erros na comunicação não foi testada. A análise exige o uso de equipamento de medição específico.

```

oper_TWI_concl=0; //trava do sistema até a conclusão da transmissão
start_bit(); //envia o Start bit. Passo (1)
passo = 1;
cont_max_erro = 0xFF;

while(oper_TWI_concl==0);/*se for crítica a espera, o programa principal pode
gerenciar esta operação*/
}

//-----
unsigned char le_DS1307(unsigned char end_leitura)
{
    //passa variáveis da função para as variáveis globais alteradas na ISR
    escrita = 0; //1 para escrita 0 para leitura
    end_escr_leit = end_leitura;

    oper_TWI_concl=0; //trava do sistema até a conclusão da transmissão
    start_bit(); //envia o Start bit. Passo (1)
    passo = 1;
    cont_max_erro=0xFF;

    while(oper_TWI_concl==0); /*se for crítica a espera, o programa principal pode
gerenciar esta operação*/
    return dado_leit;
}

//-----
ISR(TWI_vect)//Rotina de interrupção da TWI
{
    static unsigned char fim_escrita;

    switch (TWSR & 0xF8) //lê o código de resultado do TWI e executa a próxima ação
    {
        /*LEITURA E ESCRITA
        PASSO 2 <start condition transmitted>. Passo (1) concluído, executa passo (2)*/
        case (TW_START):
            TWDR = 0xD0; //envia endereço do dispositivo e o bit de escrita
            clr_start_bit(); //limpa o start bit
            passo = 2;
            break;

        /*LEITURA E ESCRITA
        PASSO 3 <SLA+W transmitted, ACK received>. Passo (2) concluído, executa passo (3)*/
        case (TW_MT_SLA_ACK):
            TWDR = end_escr_leit;//envia o endereço de escrita ou leitura
            passo=3;
            fim_escrita=0; //inicializa variável para uso na escrita, PASSO 4
            break;

        /*LEITURA E ESCRITA
        PASSO 4 <data transmitted, ACK received>. Passo (3) concluído, executa passo (4).
        Passo (4) concluído, executa passo (5) (só na escrita). O passo (4) para uma leitura é o
        reinício*/
        case (TW_MT_DATA_ACK):
            if(fim_escrita) //se o passo (4) foi concluído executa o (5), escrita
            {
                stop_bit();
                oper_TWI_concl = 1; //avisa que operação foi concluída
                break;
            }
            //envia um único dado quando for operação de escrita e depois um stop_bit()
            if(escrita)
            {
                TWDR = dado_escr;//dado para escrita no endereço de escrita
                fim_escrita = 1;//avisa que é o último dado a ser escrito
            }
    }
}

```

```

        else
            start_bit(); //envia reinício (só para operação de leitura)

        passo = 4;
        break;

/*LEITURA
PASSO 5 <repeated start condition transmitted>. Passo (4) concluído, executa o (5)*/
case (TW_REP_START):
    TWDR = 0xD1;           //Envia endereço e read bit (4)
    clr_start_bit();       //limpa start bit
    passo = 5;
    break;

/*LEITURA
PASSO 6 <SLA+R transmitted, ACK received>. Passo (5) concluído, prepara para a
execução do NACK*/
case (TW_MR_SLA_ACK) :
    TWCR &= ~(1<<TWEA); //enviará um NACK após a recepção do dado
    passo=6;
    break;

/*LEITURA
PASSO 7 <data received, NACK returned>. Passo (6) concluído, NACK recebido, executa
passo (7)*/
case (TW_MR_DATA_NACK):
    dado_leit = TWDR;     //dado lido
    stop_bit();
    oper_TWI_concl = 1; //avisa que operação foi concluída
    break;

/*TRATAMENTO DOS POSSÍVEIS ERROS
Quando um erro acontece a operação errada é repetida até funcionar ou até
que o contador para o número máximo de tentativas chegue a zero*/

default:
    cont_max_erro--;
    switch(passo)
    {
        case(1): start_bit(); break;
        case(2): TWDR = 0xD0; break;
        case(3): TWDR = end_escr_leit; break;
        case(4):
            if(escrita)
                TWDR = dado_escr;
            else
                start_bit(); //reinício
            break;
        case(5): TWDR = 0xD1; break;
        case(6): TWCR &= ~(1<<TWEA); break;
    }
    /*para saber se houve estouro na contagem ou insucesso na
       correção dos erros, basta ler cont_max_erro.*/
    if(cont_max_erro==0)
    {
        stop_bit();
        oper_TWI_concl = 1; //libera o sistema
    }
}

TWCR |= (1<<TWINT); //limpa flag de interrupção
}

```

```

//-----
//Rotinas complementares para o trabalho com o LCD
//-----
void converte2BCD(unsigned char dado, unsigned char endereco)
{
    endereco = endereco + endereco;          //2 x endereco, são dois digitos por dado
    segs[endereco] = dado & 0b00001111;      //BCD - primeiro digito - direita
    segs[endereco+1] = (dado & 0b11110000)>>4; //BCD - segundo digito - esquerda
}

//rotina para ler o dado do RTC e converter para o valor de segmento correto
//-----
void ler_convert_tudo(unsigned char ende)//lê todo o tempo do RTC e converte para o LCD
{
    unsigned char j;
    for (j=0;j<7;j++)//leitura e conversão do tempo do RTC
    {
        tempo[j] = le_DS1307(j+ende); //salva todos os dados do relógio
        converte2BCD(tempo[j], j);
    }/*j -> 0x00 segundos, 0x01 minutos, 0x02 horas, 0x03 dia semana, 0x04 dia mês,
       0x05 mês, 0x06 ano.*/
}

//-----
//endereço de ajuste e seu limite
void ajusta_tempo(unsigned char end, unsigned char limite,unsigned char ini_nr1)
{
    //ini_nr1 -> flag para avisar se o nr. começa em 1 (dia e mês)
    unsigned char aux, conta=0;

    aux = segs[end+end] + 10*segs[end+end+1];//converte 2 BCDs em 1 byte

    if (aux<limite)
        aux++; //incremento no tempo
    else
        aux = 0;

    //converte aux para 2 BCD
    if (aux<10)
        tempo[end] = aux;
    else
    {
        do
        {
            aux = aux - 10;
            conta++;
        }while(aux>9);
    }

    if(ini_nr1==1)
    {
        if (aux==0 && conta==0)//se for dia e mês não pode mostrar o zero
            aux = 1;
    }

    tempo[end] = aux + (conta<<4);//monta o tempo ajustado - 2 BCDs
    escreve_DS1307(end,tempo[end]); //altera o valor no RTC
}

//-----
// Rotina que chama o ajuste_tempo conforme seleção de botao
//-----
void ajusta_tempo2(unsigned char qual)
{
    switch(qual)//qual ajuste será efetuado
    {
        case 0: break;
        case 1: ajusta_tempo(0x02, 23,0); break;//ajusta as horas (0x02) e seu limite (23)
}

```

```

        case 2: ajusta_tempo(0x01, 59,0);    break;//ajuste dos minutos
        case 3: escreve_DS1307(0x00, 0x00); break;//só zera os segundos
        case 4: ajusta_tempo(0x04, 31,1);   break;//ajuste dos dias
        case 5: ajusta_tempo(0x05, 12,1);   break;//ajuste dos meses
        case 6: ajusta_tempo(0x06, 99,0);   break;//ajuste do ano
        case 7: ajusta_tempo(0x03, 7,1);    break;
    } //switch(count)
}

//-----
void mostra_tempo()//coloca os dados do RTC no formato padrão no LCD
{
    //mostrar tempo no LCD -> 1a linha horas
    cmd_LCD(0x81,0);
    cmd_LCD(segs[5]+48,1);    //horas
    cmd_LCD(segs[4]+48,1);
    cmd_LCD(0x84,0);
    cmd_LCD(segs[3]+48,1);    //minutos
    cmd_LCD(segs[2]+48,1);
    cmd_LCD(0x87,0);          //espaço em branco
    cmd_LCD(segs[1]+48,1);    //segundos
    cmd_LCD(segs[0]+48,1);
    //mostrar tempo no LCD -> 2 linha calendário
    cmd_LCD(0xC1,0);
    cmd_LCD(segs[9]+48,1);    //dia do mês
    cmd_LCD(segs[8]+48,1);
    cmd_LCD(0xC4,0);          //espaço em branco
    cmd_LCD(segs[11]+48,1);   //mês
    cmd_LCD(segs[10]+48,1);
    cmd_LCD(0xC7,0);          //espaço em branco
    cmd_LCD(segs[13]+48,1);   //ano
    cmd_LCD(segs[12]+48,1);

    switch(segs[6])//dia da semana
    {
        case 1: escr[0] = 'D'; escr[1] = 'o'; escr[2] = 'm'; break;
        case 2: escr[0] = 'S'; escr[1] = 'e'; escr[2] = 'g'; break;
        case 3: escr[0] = 'T'; escr[1] = 'e'; escr[2] = 'r'; break;
        case 4: escr[0] = 'Q'; escr[1] = 'u'; escr[2] = 'a'; break;
        case 5: escr[0] = 'Q'; escr[1] = 'u'; escr[2] = 'i'; break;
        case 6: escr[0] = 'S'; escr[1] = 'e'; escr[2] = 'x'; break;
        case 7: escr[0] = 'S'; escr[1] = 'a'; escr[2] = 'b'; break;
        default: escr[0] = 'x'; escr[1] = 'x'; escr[2] = 'x'; break;
    } //switch

    cmd_LCD(0xCC,0);
    escreve_LCD(escr);//mostra dia da semana
}

//-----
void mostraPontos(unsigned char desliga)//coloca pontos e barra no display e limpa
//algumas posições
{
    if (desliga==1)
    {
        cmd_LCD(0x80,0);cmd_LCD(0x20,1); //limpa antes da hora
        cmd_LCD(0x83,0);cmd_LCD(0x3A,1); //dois pontos depois das horas
        cmd_LCD(0x86,0);cmd_LCD(0x3A,1); //dois pontos depois dos segundos
        cmd_LCD(0xC0,0);cmd_LCD(0x20,1); //limpa antes do dia do mês
        cmd_LCD(0xC3,0);cmd_LCD(0x2F,1); //barra inclinada depois do dia do mês
        cmd_LCD(0xC6,0);cmd_LCD(0x2F,1); //barra inclinada antes do ano
        cmd_LCD(0xCB,0);cmd_LCD(0x20,1); //limpa antes do dia da semana
    }
}

```

```

void alerta_display(unsigned char qual)//seta no display para o ajuste
{
    switch(qual)
    {
        case 1: cmd_LCD(0x80,0); break;      //horas
        case 2: cmd_LCD(0x83,0); break;      //minutos
        case 3: cmd_LCD(0x86,0); break;      //segundos
        case 4: cmd_LCD(0xC0,0); break;      //dia mês
        case 5: cmd_LCD(0xC3,0); break;      //mês
        case 6: cmd_LCD(0xC6,0); break;      //ano
        case 7: cmd_LCD(0xCB,0); break;      //dia da semana
    }//switch(count)

    //este if é só para corrigir a seta na última pressionada do botao SELECAO, após o dia.
    if(cont!=0)
        cmd_LCD(0x7E,1);//-> seta para ajuste
}
//-----

```

LCD.h e LCD.c (vistos no capítulo 5 – com a definição #define conv_ascii 0)

Um resultado prático para o programa supracitado é apresentado na fig. 16.17.



Fig. 16.17 - Resultado prático para teste do programa de controle do TWI empregando máquina de estados e o DS1307.

Exercícios:

- 16.1** – Elaborar um programa para ler e escrever em um DS1307 (*Real Time Clock*) empregando o módulo TWI do ATmega328, conforme circuito da fig. 16.16. Altere o programa apresentado para que ele não utilize a interrupção do TWI e faça a atualização do LCD a cada 1 segundo utilizando o sinal proveniente do pino SOUT do DS1307.
- 16.2** – Baseado no programa apresentado anteriormente, altere as rotinas de trabalho com o DS1307 para a leitura e escrita sequencial de vários bytes.
-

17. COMUNICAÇÃO 1 FIO POR SOFTWARE

Neste capítulo, é descrita a técnica de comunicação baseada no emprego de somente um fio para a transmissão de dados. Essa técnica permite a diminuição do número necessário de pinos de I/O do microcontrolador e da complexidade do hardware necessário para realizar uma tarefa específica. Como a maioria dos microcontroladores, o ATmega não dispõe de hardware dedicado para a comunicação 1 fio, sendo que essa deve ser totalmente realizada via programação.

A comunicação 1 fio, conhecida por *1 wire*, foi desenvolvida pela antiga Dallas (adquirida pela MAXIM) e permite, através de uma única via, conectar diferentes circuitos integrados dedicados. A comunicação é realizada em um único fio, utilizando espaços de tempos adequados para cada nível do sinal digital. Os CIs 1 *wire* podem ser empregados com o uso de 3 vias: VCC, GND e sinal; ou ainda, apenas duas vias: GND e sinal, quando se emprega a alimentação parasita⁴¹.

Os componentes 1 *wire* possuem as seguintes características:

- 1 única via para enviar e receber dados, sem a necessidade de uma via de *clock*.
- Cada CI possui um único número de identificação gravado na fabricação.
- A alimentação pode ser feita pela via de sinal (alimentação parasita).
- Suportam vários dispositivos na mesma linha (fig. 17.1). O dispositivo de controle é o mestre e os demais são os escravos.

⁴¹ Na alimentação parasita, o CI é alimentado pela via de dados, a qual deve ser mantida na tensão de alimentação por um período de tempo suficiente para carregar um capacitor interno, permitindo o funcionamento do CI quando a via de dados estiver em nível lógico zero. Nesse modo de alimentação, o pino de VCC deve ser ligado ao terra do circuito.

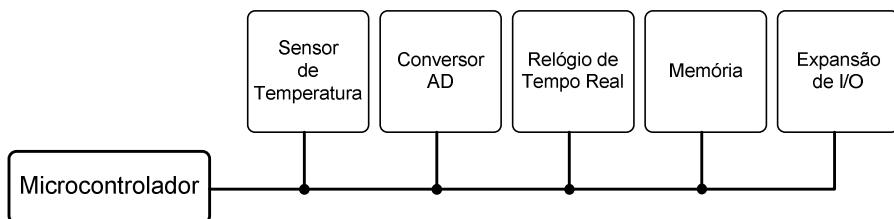


Fig. 17.1 – Conexão 1 fio para vários dispositivos. O microcontrolador é o mestre.

Os circuitos integrados disponíveis com a comunicação 1 *wire* são: memórias, sensores de temperatura e chaves de temperatura, conversores AD, relógios de tempo real, protetores de baterias, conversores I2C-1 *wire* (expansão de I/Os), entre outros.

A seguir, são apresentados alguns diagramas temporais para ilustrar o funcionamento do protocolo 1 *wire* aplicado ao DS18S20 (sensor de temperatura). O detalhamento completo é encontrado no manual do fabricante. Na fig. 17.2, é apresentado o protocolo para sua inicialização. O dispositivo de controle (mestre) transmite um pulso de inicialização ($> 480\mu s$) e, em seguida, libera a linha e vai para o modo de recepção. O barramento 1 *wire* é colocado no estado lógico alto através de um resistor externo de *pull-up* de $4,7\text{ k}\Omega$. Depois de detectar a borda de subida no pino de dados, o DS18S20 aguarda de 15 a 60 μs e, então, transmite o pulso de presença (60 a 240 μs). Um pulso de inicialização deve ser sempre enviado antes de um comando.

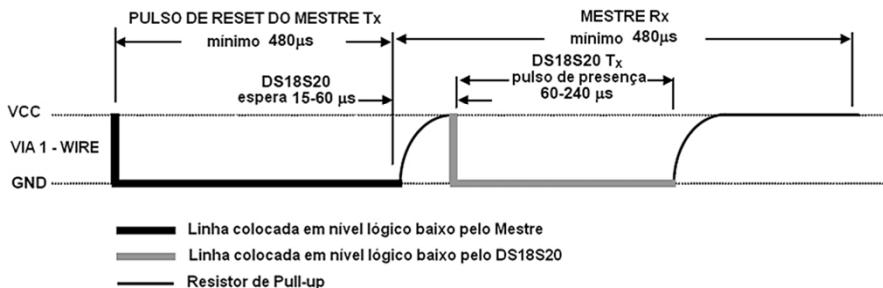


Fig. 17.2 – Diagrama temporal de inicialização do protocolo 1-wire.

O protocolo para a leitura e a escrita do DS18S20 é apresentado na fig. 17.3. Na escrita, a borda de descida na linha de dados sincroniza o dispositivo escravo com o mestre disparando um circuito interno de atraso. Esse atraso determina quando o escravo lerá a linha de dados (de 15 µs a 60 µs). Na leitura, a borda de descida na linha de dados sincroniza os dados do escravo com o mestre, novamente disparando um circuito de atraso interno. Se a transmissão for um ‘0’, o circuito de atraso determina quanto tempo o dispositivo irá manter a linha de dados em nível lógico baixo, substituindo o ‘1’ gerado, pela tensão do resistor de *pull-up*. Se for para transmitir ‘1’, o dispositivo deixará inalterado o intervalo de tempo para a leitura de dados.

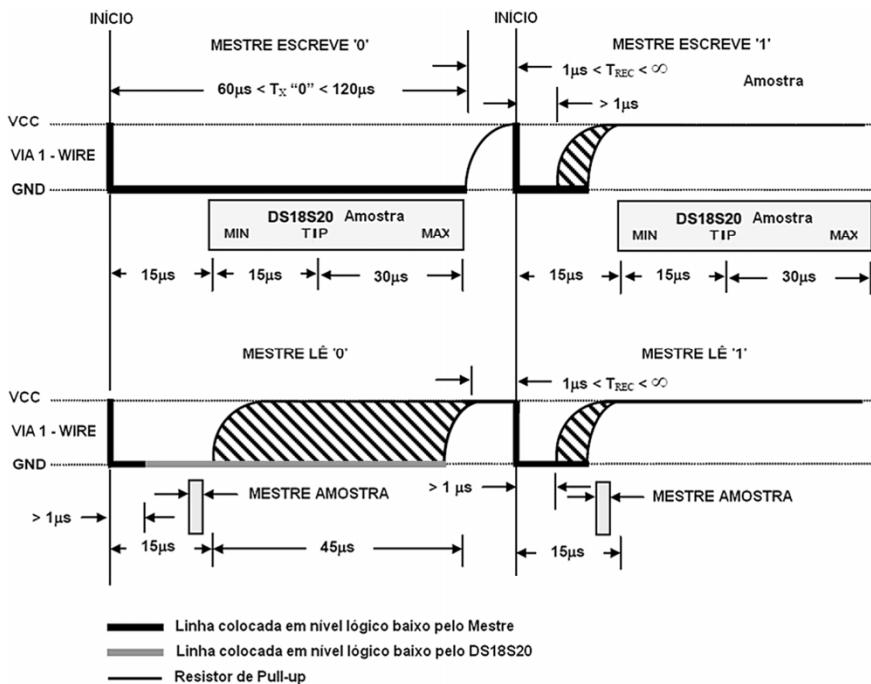


Fig. 17.3 – Diagrama temporal para leitura e escrita em um DS18S20.

Cada dispositivo 1 wire contém um código único de 64 bits armazenado em ROM. Os 8 bits menos significativos do código ROM contém o código da família, os próximos 48 bits contêm um número de série único. Os 8 bits mais significativos contêm um byte de verificação de redundância cíclica (CRC) que é calculado a partir dos primeiros 56 bits do código de ROM (ver a fig. 17.4). Como o código é único, é possível ter vários dispositivos iguais no mesmo barramento, tais como vários DS18S20.

8-BIT CRC		48-BIT Número Serial		8-BIT Código da Família	
MSB	LSB	MSB	LSB	MSB	LSB

Fig. 17.4 – Código de 64 bits dos dispositivos 1 wire.

17.1 SENSOR DE TEMPERATURA DS18S20

O DS18S20 é um sensor digital de 9 bits de resolução⁴² com 0,5°C de precisão, que emprega o protocolo 1-wire e permite múltiplos sensores na mesma linha. Sua faixa de medição vai de -55 °C até +125 °C e o tempo de conversão é de 750 ms; também possui sinalização de alarme para a temperatura. É encontrado nos encapsulamentos TO-92 ou SMD. Alguns exemplos de valores de temperatura convertidos por ele são apresentados na tab. 17.1.

Tab. 17.1 – Exemplos de conversões do DS18S20.

Temperatura °C	Saída Digital Binária	Saída Digital Hexadecimal
+85,0	0000 0000 1010 1010	0x00AA
+25,0	0000 0000 0011 0010	0x0032
+0,5	0000 0000 0000 0001	0x0001
0	0000 0000 0000 0000	0x0000
-0,5	1111 1111 1111 1111	0xFFFF
-25,0	1111 1111 1100 1110	0xFFCE
-55,0	1111 1111 1001 0010	0xFF92

Na tab. 17.2, é apresentada a memória interna (RAM) do DS18S20. A memória também possui 2 bytes de EEPROM, que não sendo utilizados para o ajuste do alarme de temperatura podem ser de uso geral. Os bytes 6 e 7 podem ser empregados para calcular uma temperatura com mais de 9 bits de resolução (ver o manual do fabricante). Os valores do byte 2 e 3 dependem dos valores armazenados na EEPROM.

Tab. 17.2 – Memória do DS18S20 (valores na energização).

Bytes	Saída Digital Binária	EEPROM	
0	Temperatura LSB (0xAA)		
1	Temperatura MSB (0x00)		
2	TH* ou byte 1 do usuário	↔	TH* ou byte 1 do usuário
3	TL* ou byte 2 do usuário	↔	TL* ou byte 2 do usuário
4	Reservado (0xFF)		
5	Reservado (0xFF)		
6	Count Remain (0x0C)		
7	Count Per °C (0x10)		
8	CRC		

* TH e TL registradores de ajuste de disparo do alarme.

⁴² O DS18B20 possui resolução ajustável de 9 a 12 bits.

COMANDOS DE ROM

Os comandos de ROM podem ser utilizados pelo dispositivo mestre após a detecção do pulso de presença. Servem para operar com o código único de 64 bits caso exista mais de um dispositivo escravo no barramento. Esses comandos permitem determinar quantos e que tipos de dispositivos estão presentes e se algum deles tem alguma condição de alarme ativa.

- Procura de ROM [0xF0] - Quando o sistema é energizado, o mestre deve identificar o código ROM de todos os escravos, determinando o número deles e seu tipo. O código é obtido através do processo de eliminação, que requer ao mestre a execução de um ciclo de procura de ROM. Isto é, o código de procura é seguido pela troca de dados e deve ser feito até a identificação de todos os escravos. Se só existir um escravo, o comando de procura deve ser substituído pelo comando de leitura da ROM.
- Leitura da ROM [0x33] – Este comando só pode ser usado quando existir apenas um escravo. Permite ao mestre ler o código de 64 bits do escravo.
- Igualdade da ROM [0x55] – Este comando, seguido pelo código de 64 bits, permite ao mestre endereçar um escravo específico. Só o escravo com o endereço igual ao transmitido irá responder.
- Pular ROM [0xCC] – Pula o processo de leitura da ROM e permite ao mestre mandar todos os dispositivos do barramento executarem uma conversão de temperatura após o comando de conversão (0x44), sem a necessidade de transmissão do código de 64 bits.
- Procura de Alarme [0xEC] – Este comando é similar a procura de ROM, com exceção de que somente o escravo com a sinalização de alarme irá responder. Permite ao mestre saber se algum escravo sinalizou uma condição de alarme, isto é, a temperatura medida está fora de uma faixa estabelecida.

COMANDOS DE FUNÇÕES

Os comandos de função são utilizados pelo mestre após os comandos de ROM para determinar o que deve ser feito pelo escravo.

- Converte Temperatura [0x44] – Inicia uma conversão de temperatura. Após a conversão, o resultado é armazenado em um registrador de 2 bytes na memória do DS18S20 que, então, retorna ao estado de espera.
- Escrita [0x4E] – Permite ao mestre escrever 2 bytes de dados no DS18S20, o primeiro é escrito no registrador TH e o segundo no TL. Primeiro devem ser transmitidos os bits menos significativos.
- Leitura [0xBE] – Permite ao mestre ler o conteúdo da memória do DS18S20. A transferência de dados começa com o bit menos significativo do byte 0 e continua através de toda a memória até que o nono byte seja lido. O mestre pode inicializar o barramento a qualquer momento para terminar a leitura no instante desejado.
- Copia memória [0x48] – Copia o conteúdo dos registradores TH e TL (bytes 2 e 3) para a EEPROM.
- Re-chamada E2 [0xB8] – Copia o valor de disparo do alarme (TH e TL) da EEPROM e os escreve nos bytes de dados 2 e 3.
- Leitura da fonte de energia [0xB4] – Este comando, seguido por uma leitura da resposta do DS18S20, permite determinar se este está usando alimentação parasita.

Nas tabs. 17.3-5, são apresentados exemplos das sequências de comandos para o trabalho com o DS18S20.

Tab. 17.3 – Exemplo1: existem múltiplos DS18S20 e eles empregam alimentação parasita. O mestre inicializa a conversão de um específico DS18S20, lê sua memória e calcula o CRC para verificação.

Modo Mestre	Dado (primeiro o LSB)	Comentários
Tx	Reset	O mestre executa o pulso de <i>reset</i> .
Rx	Presença	O DS18S20 responde com um pulso de presença.
Tx	0x55	O mestre envia o comando de igualdade da ROM.
Tx	Código de 64 bits	O mestre envia o código do DS18S20 em questão.
Tx	0x44	O mestre pede para começar a conversão.
Tx	A linha DQ é mantida em VCC	O mestre aplica tensão na linha DQ durante o tempo de conversão.
Tx	Reset	O mestre executa o pulso de <i>reset</i> .
Rx	Presença	O DS18S20 responde com um pulso de presença.
Tx	0x55	O mestre envia o comando de igualdade da ROM.
Tx	Código de 64 bits	O mestre envia o código do DS18S20 em questão.
Tx	0xBE	O mestre pede o comando para a leitura dos dados.
Rx	9 bytes de dados	O mestre lê toda a memória e calcula o CRC. Se houver uma igualdade o mestre continua, senão, a operação de leitura deve ser repetida.

Tab. 17.4 – Exemplo 2: existe somente um DS18S20 com alimentação parasita. O mestre escreve em TH e TL, lê a memória e calcula o CRC para verificação. Então, copia TH e TL para a EEPROM

Modo Mestre	Dado (primeiro o LSB)	Comentários
Tx	Reset	O mestre executa o pulso de <i>reset</i> .
Rx	Presença	O DS18S20 responde com um pulso de presença.
Tx	0xCC	O mestre pede o comando para pular a ROM.
Tx	0x4E	O mestre pede para escrever.
Tx	2 bytes de dados	O mestre envia dois bytes para TH e TL.
Tx	Reset	O mestre executa o pulso de <i>reset</i> .
Rx	Presença	O DS18S20 responde com um pulso de presença.
Tx	0xCC	O mestre pede o comando para pular a ROM.
Tx	0xBE	O mestre pede o comando para a leitura dos dados.
Rx	9 bytes de dados	O mestre lê toda a memória e calcula o CRC. Se houver uma igualdade o mestre continua, senão uma nova operação de leitura deve ser repetida.
Tx	Reset	O mestre executa o pulso de <i>reset</i> .
Rx	Presença	O DS18S20 responde com um pulso de presença.
Tx	0xCC	O mestre pede o comando para pular a ROM.
Tx	0x48	O mestre pede para copiar TH e TL para a EEPROM
Tx	A linha DQ é mantida em VCC	O mestre aplica tensão na linha DQ durante o tempo de conversão.

Tab. 17.5 – Exemplo 3: existe somente um DS18S20 com alimentação parasita. O mestre manda iniciar uma conversão e lê os bytes que lhe interessam (modo mais simples de operação).

Modo Mestre	Dado (primeiro o LSB)	Comentários
Tx	Reset	O mestre executa o pulso de <i>reset</i> .
Rx	Presença	O DS18S20 responde com um pulso de presença.
Tx	0xCC	O mestre pede o comando para pular a ROM.
Tx	0x44	O mestre pede para começar a conversão.
Tx	A linha DQ é mantida em VCC	O mestre aplica tensão na linha DQ durante o tempo de conversão.
Tx	Reset	O mestre executa o pulso de <i>reset</i> .
Rx	Presença	O DS18S20 responde com um pulso de presença.
Tx	0xCC	O mestre pede o comando para pular a ROM.
Tx	0xBE	O mestre pede o comando para a leitura dos dados.
Rx	2 ou os 9 bytes de dados	Os dois primeiros bytes são a temperatura. Se desejado podem ser lidos os demais bytes, que incluem, entre outros, os 2 bytes da EEPROM e a identificação do componente (CRC).
Tx	Reset	O mestre executa o pulso de <i>reset</i> após ler o número desejado de bytes, se for só temperatura, após 2 bytes.

Quando a alimentação parasita não é empregada, basta esperar o tempo da conversão sem manter a linha DQ em VCC. Para o cálculo do CRC, deve-se consultar o manual do fabricante.

A seguir, é apresentado o código de leitura para o circuito da fig. 17.5 com a alimentação parasita do DS18S20 e a sequência mais simples de trabalho, mostrado na tab. 17.5 (exemplo 3). A informação de temperatura é transmitida ao computador (ver o capítulo 15).

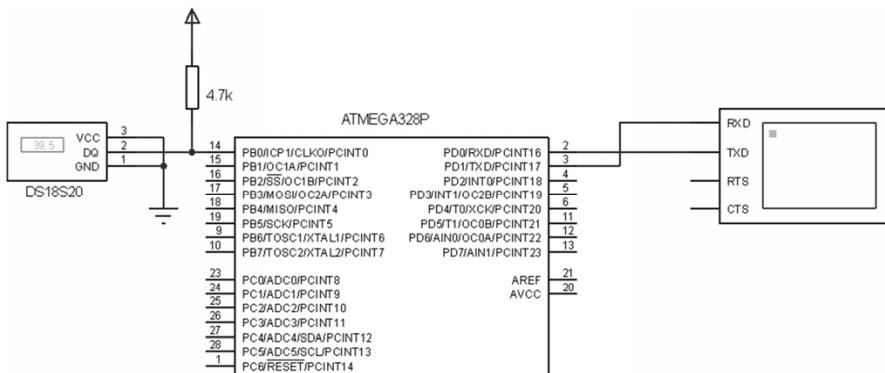


Fig. 17.5 – Circuito de teste para o programa do exemplo 3 (tab. 17.5).

DS18S20_1wire.c (programa principal)

```
//===================================================================== //
//      Programa para a leitura de temperatura de um DS18S20          //
//      Envio dos dados ao computador                                     //
//===================================================================== //
#include "def_principais.h"
#include "USART.h"
#include "_1wire.h"

const char msg[] PROGMEM = "Sensor de Temperatura DS18S20\n\0";
unsigned char digitos[tam_vetor];
//-----
int main()
{
    char sinal;
    unsigned char temp_LSB, temp_MSB;
    unsigned int valor_temp;

    USART_Inic(MYUBRR);
    escreve_USART_Flash(msg);

    while(1)
    {
        reset_1w(); //reset do sensor (a resposta de presença é retornada mas não avaliada).
        escreve_byte_1w(0xCC); //comando para pular ROM (só 1 dispositivo no barramento).
        escreve_byte_1w(0x44); //manda iniciar a conversão
        alimenta_1w(); //alimentação parasita (quando usar comentar a linha abaixo)
        //_delay_ms(750); //sem alimentação parasita (quando usar comentar a linha acima)
        reset_1w();
        escreve_byte_1w(0xCC);
        escreve_byte_1w(0xBE); //avisa que quer ler a memória

        temp_LSB = le_byte_1w(); //só interesse em ler os dois bytes da temperatura
        temp_MSB = le_byte_1w();
    }
}
```

```

if (temp_MSB>0)           //verifica se a temperatura é negativa
{
    temp_LSB = 256 - temp_LSB;//converte número negativo para positivo
    sinal = '-';//sinaliza o caractere de sinal da temperatura para o LCD
}
else
    sinal='+';

//conversão do número lido para um valor inteiro sem vírgula (temp_LSB*10/2)
valor_temp = 5*temp_LSB;//retirar a vírgula do número e converter, LSB vale 0,5°C
                           //Para uso do DS18B20 a formula deve ser alterada.
ident_num(valor_temp,digitos);
USART_Transmite(sinal);   //envio dos dado ao computador
USART_Transmite(digitos[3]);
USART_Transmite(digitos[2]);
USART_Transmite(digitos[1]);
USART_Transmite(',');
USART_Transmite(digitos[0]);
USART_Transmite(176);     //símbolo 'o'
USART_Transmite('C');
USART_Transmite('\n');

}

}

//-----

```

_1wire.h (arquivo de cabeçalho do _1wire.c)

```

ifndef __1WIRE_H
#define __1WIRE_H

#include "def_principais.h"

#define DDRx    DDRB      //define o DDR do pino DQ
#define PORTx  PORTB      //define o PORT do pino DQ
#define PINx   PINB      //define o PIN do pino DQ
#define DQ     PB0       //pino para a comunicação

#define DQ_saida()      set_bit(DDRx,DQ)
#define DQ_entrada()    clr_bit(DDRx,DQ)
#define clr_DQ()        clr_bit(PORTx,DQ)
#define set_DQ()        set_bit(PORTx,DQ)
#define tst_DQ()        tst_bit(PINx,DQ)

unsigned char reset_1w();
void alimenta_1w();
unsigned char le_bit_1w();
void escreve_bit_1w(unsigned char bit_1w);
unsigned char le_byte_1w();
void escreve_byte_1w(unsigned char dado);

#endif

```

_1wire.c (arquivo com as funções para trabalho com o 1 wire)

```
//===================================================================== //
//          ROTINAS PARA A COMUNICAÇÃO 1 WIRE                         //
//===================================================================== //
#include "_1wire.h"
//-----
unsigned char reset_1w()//inicializa os dispositivos no barramento
{
    DQ_saida();      //DQ como saída
    clr_DQ();        //DQ em nível zero por 480us
    _delay_us(480);

    DQ_entrada();    //DQ como entrada, o resistor de pull-up mantém DQ em nível alto
    _delay_us(60);

    if(tst_DQ())    //se não detectou a presença já retorna 1
        return 1;

    _delay_us(420); //o pulso de presença pode ter 240 us
    return 0;        //retorna 0 para indicar sucesso
}
//-----
void alimenta_1w() //força o barramento em nível alto.
{
    DQ_saida();      //utilizado com dispositivos alimentados no modo parasita
    set_DQ();
    _delay_ms(750);
    DQ_entrada();   //pull-up externo
}
//-----
unsigned char le_bit_1w() //lê um bit do barramento
{
    unsigned char dado;

    DQ_saida();
    clr_DQ();
    _delay_us(2);
    DQ_entrada();
    _delay_us(15);    //aguarda o dispositivo colocar o dado na saída
    dado = tst_DQ();
    _delay_us(50);
    return (dado);   //retorna o dado
}
//-----
void escreve_bit_1w(unsigned char bit_1w)//escreve um bit no barramento
{
    DQ_saida();
    clr_DQ();
    _delay_us(2);

    if(bit_1w)
        set_DQ();

    _delay_us(120);
    DQ_entrada();
}
//-----
```

```

unsigned char le_byte_1w() //lê um byte do barramento
{
    unsigned char i, dado = 0;
    for (i=0;i<8;i++) //lê oito bits iniciando pelo bit menos significativo
    {
        if (le_bit_1w())
            set_bit(dado,i);
    }
    return (dado);
}
//-----
void escreve_byte_1w(unsigned char dado) //escreve um byte no barramento
{
    unsigned char i;
    for (i=0; i<8; i++) //envia o byte iniciando com o bit menos significativo
        escreve_bit_1w(tst_bit(dado,i)); //escreve o bit no barramento
}
//-----

```

O arquivo **def_principais.h** é igual ao apresentado anteriormente. **USART.h** e **USART.c** foram vistos no capítulo 15.

Exercícios:

17.1 – Consulte o manual do DS18S20 para saber como o CRC é gerado e como deve ser calculado. Utilize essa informação para resolver o exercício 17.2.

17.2 – Elaborar um programa para trabalhar com dois sensores de temperatura DS18S20, conforme fig. 17.6. Antes de colocá-los juntos no barramento 1 fio é necessário ler individualmente o código único de 64 bits de cada sensor. Apresente o valor do CRC de cada DS18S20 em uma linha do LCD 16×2 em conjunto com sua temperatura.

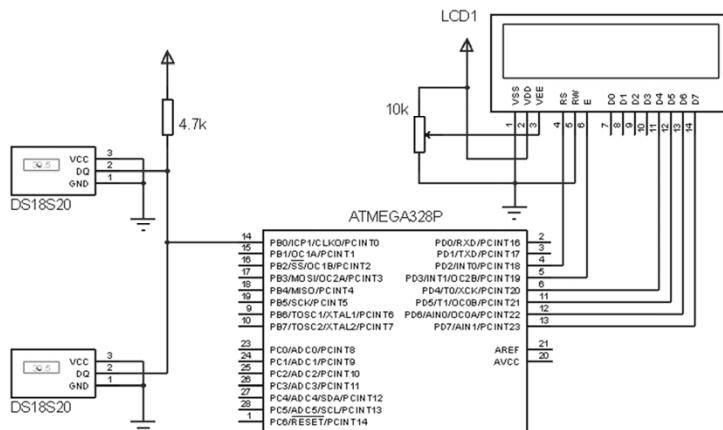


Fig. 17.6 – Circuito com dois DS18S20.

18. COMPARADOR ANALÓGICO

Neste capítulo, é explicado o uso do comparador analógico do ATmega328 e apresentadas técnicas para o seu emprego na leitura de grandezas físicas, tais como: resistências e capacitâncias. Ainda, apresenta-se uma técnica para a leitura de múltiplas teclas.

O comparador analógico do ATmega328 utiliza um amplificador operacional para comparar os valores de entrada do pino positivo AIN0 e do pino negativo AIN1. Quando a tensão no pino positivo é maior que a tensão no pino negativo, a saída do comparador analógico é ativa (vai a 1). Se necessário, essa saída pode ser ajustada para disparar um evento de captura do Temporizador/Contador 1. A interrupção do comparador pode ser disparada na transição dos sinais de entrada: baixo para alto, alto para baixo ou em qualquer transição. O diagrama em blocos do comparador é apresentado na fig. 18.1.

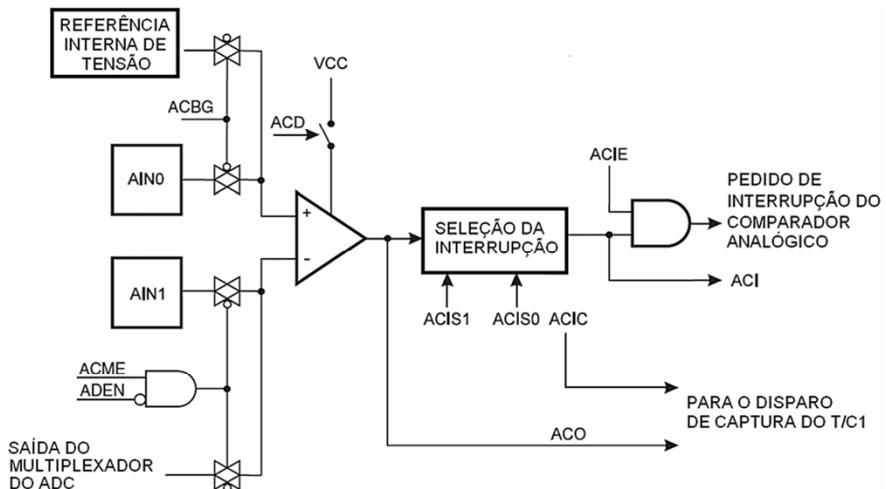


Fig. 18.1 – Diagrama do comparador analógico.

REGISTRADORES

ADCSR – ADC Control and Status Register B

Bit	7	6	5	4	3	2	1	0
ADCSR B	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0
Lê/Escrive	L	L/E	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 6 – ACME – Analog Comparator Multiplexer Enable

Quando este bit é colocado em nível lógico 1 com o conversor AD desligado (ADEN no ADCSRA é zero), o multiplexador do AD seleciona a entrada negativa do comparador analógico. Quando colocado em nível lógico zero, AIN1 é aplicado na entrada negativa do comparador.

ACSR – Analog Comparator Control and Status Register

Bit	7	6	5	4	3	2	1	0
ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACISO
Lê/Escrive	L/E	L/E	L	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	N/A ⁴³	0	0	0	0	0

Bit 7 – ACD – Analog Comparator Disable

Quando este bit for 1, a alimentação do comparador analógico é desligada. Este bit pode ser empregado a qualquer momento para desligar o comparador e economizar energia. Quando ACD for alterado, a interrupção do comparador deve estar desabilitada, caso contrário, pode ocorrer uma interrupção.

Bit 6 – ACBG – Analog Comparator Bandgap Select

Quando este bit for 1, uma referência fixa de tensão substitui a entrada positiva do comparador analógico (aprox. 1,1V). Quando ACBG=0, AINO é aplicado na entrada positiva do comparador. Quando for utilizada a referência fixa, existe um tempo para a sua estabilização antes que se possa fazer uma leitura correta.

Bit 5 – ACO – Analog Comparator Output

A saída do comparador analógico é sincronizada e, então, diretamente conectada ao ACO. A sincronização introduz um atraso de 1 a 2 ciclos de *clock*.

Bit 4 – ACI – Analog Comparator Interrupt Flag

Este bit é ativo por hardware quando houver um evento de disparo da interrupção do comparador.

Bit 3 – ACIE – Analog Comparator Interrupt Enable

ACIE=1 habilita a interrupção do comparador em conjunto com o bit I do SREG.

Bit 2 – ACIC – Analog Comparator Input Capture Enable

Quando ativo, habilita a função de entrada de captura do Temporizador/Contador 1.

⁴³ N/A = Não Aplicável ou não disponível (*Not Applicable or Not Available*).

Bits 1:0 – ACIS1:0 – Analog Comparator Interrupt Mode Select

Estes bits definem qual evento irá disparar a interrupção do comparador analógico (tab. 18.1). Quando estes bits forem alterados, a interrupção deve estar desabilitada para evitar uma interrupção indesejada.

Tab. 18.1 – Ajuste dos bits ACIS1 e ACISO.

ACIS1	ACISO	Modo de interrupção
0	0	Subida ou descida do sinal de comparação.
0	1	Reservado.
1	0	Borda de descida do sinal de comparação.
1	1	Borda de subida do sinal de comparação.

DIDR1 – Digital Input Disable Register 1

Bit	7	6	5	4	3	2	1	0
DIDR1	-	-	-	-	-	-	AIN1D	AINOD
Lê/Escrve	L	L	L	L	L	L	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bits 1:0 – AIN1D:0D – AIN1, AINO Digital Input Disable

Estes bits definem se os pinos do comparador analógico serão entradas digitais. Devem ser colocados em 1 para o emprego do comparador e em zero para se utilizar normalmente os pinos como I/O.

É possível multiplexar a entrada negativa do comparador analógico com os pinos do ADC (ADC7..0), ou seja, o pino AIN1 do comparador pode ser substituído por qualquer pino do ADC. Se o bit de habilitação do multiplexador do comparador analógico (ACME no ADCSRB) estiver ativo e o ADC desligado (ADEN no ADCSRA é zero), os bits MUX2..0 no registrador ADMUX selecionam um pino do ADC para substituir o terminal negativo do comparador. Se ACME estiver limpo ou ADEN ativo, o pino AIN1 será a entrada negativa do comparador analógico (ver a tab. 18.2).

Tab. 18.2 – Multiplexação da entrada negativa do comparador analógico.

ACME	ADEN	MUX2..0	Entrada Negativa para o Comparador Analógico
0	x	xxx	AIN1
1	1	xxx	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6 (somente nos encapsulamentos TQFP e QFN/MLF)
1	0	111	ADC7 (somente nos encapsulamentos TQFP e QFN/MLF)

Na fig. 18.2, é apresentado um circuito exemplo onde se emprega o comparador analógico do ATmega328 para ligar um LED de acordo com a luminosidade ambiente. O dispositivo sensor é um LDR⁴⁴. Quando a luminosidade diminui, a resistência do LDR aumenta. Assim, com o divisor de tensão empregado no circuito da fig. 18.2, quando houver pouca luminosidade a tensão no terminal positivo do comparador (AIN0) será superior à tensão no terminal negativo (AIN1) e o comparador do ATmega irá alterar o estado de sua saída. O potenciômetro permite o ajuste da sensibilidade do circuito⁴⁵. O programa de teste encontra-se na sequência.

⁴⁴ O LDR (*Light Dependent Resistor*) é um foto resistor empregado como sensor de luminosidade: quanto maior a quantidade de luz que o atinge, menor é a sua resistência.

⁴⁵ Se o potenciômetro for considerado como duas resistências separadas no ponto de ajuste, o circuito de entrada para o comparador é similar a chamada Ponte de Wheatstone. Com a variação de temperatura os valores das resistências nessa ponte tendem a mudar. Entretanto, a variação percentual é praticamente a mesma nos seus resistores, mantendo estável a diferença de potencial dos pontos medidos.

No circuito da Fig. 18.2, dependendo da aplicação, pode ser empregado um resistor em série com o potenciômetro para melhorar a sensibilidade do circuito. Nesse caso, o potenciômetro deve ser configurado como resistor variável e o sinal de tensão é obtido na conexão do potenciômetro com esse novo resistor.

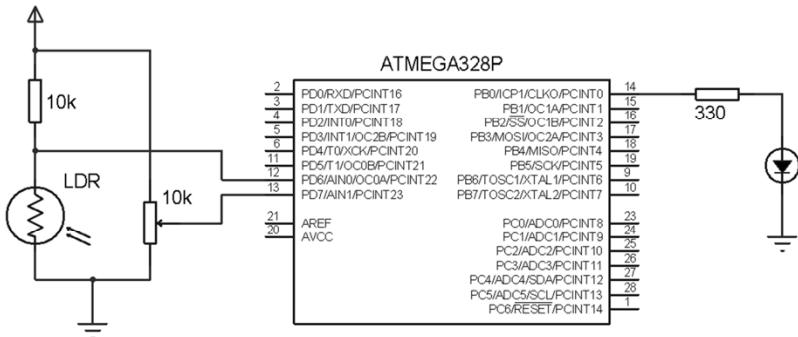


Fig. 18.2 – Sensor de luminosidade empregando um LDR e o comparador analógico do ATmega328.

Comp_analog_LDR.c

```

//=====
// SENSOR DE LUZ EMPREGANDO UM LDR
//=====

#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>

#define set_bit(Y,bit_x) (Y|=(1<<bit_x))      //ativa bit
#define clr_bit(Y,bit_x) (Y&=~(1<<bit_x))    //limpa bit
#define tst_bit(Y,bit_x) (Y&(1<<bit_x))     //testa bit

#define LED PB0

int main()
{
    DDRB = 0x01;           //pino do LED como saída
    DDRD = 0x00;           //PORT do comparador como entrada
    PORTD = 0xFF;          //habilita pull-ups
    PORTB = 0xFE;          //apaga LED e habilita pull-ups

    DIDR1 = 0b00000011;   //desabilita as entradas digitais nos pinos AIN0 e AIN1
    ACSR = 1<<ACIE;       //habilita interrup. por mudança de estado na saída do comparador
    sei();                 //habilita a chave geral de interrupções

    while(1);
}

//-----
//Interrupção do Comparador Analógico
//-----
ISR(ANALOG_COMP_vect)//O LED ficará ligado quando houver pouca luz (tensão maior no
{                      //terminal positivo do comparador)

    if(tst_bit(ACSR,AC0))//verifica qual mudança ocorreu na saída do comparador
        set_bit(PORTB,LED);
    else
        clr_bit(PORTB,LED);
}
//=====

```

18.1 MEDINDO RESISTÊNCIAS E CAPACITÂNCIAS

O comparador analógico pode ser empregado para medir resistências e capacitâncias empregando um circuito RC, onde o tempo de carga e/ou descarga do capacitor será medido em relação a uma tensão de referência empregando-se o modo de captura do TC1.

RESISTÊNCIA⁴⁶

O circuito da fig. 18.3 ilustra um circuito para a medição precisa de uma resistência desconhecida R_x , com base no valor conhecido de um resistor de referência R_{ref} e um capacitor de valor exato conhecido.

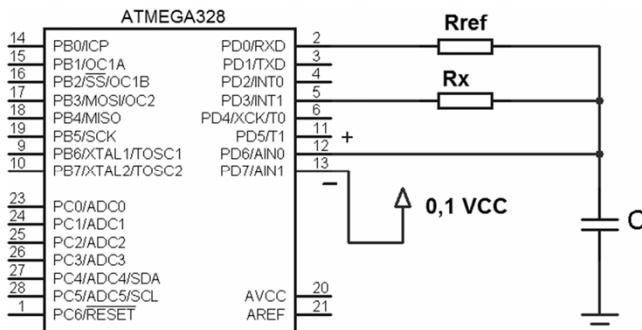


Fig. 18.3 – Circuito para medir uma resistência desconhecida R_x .

Para o funcionamento, o comparador analógico deve disparar a captura do TC1 quando a tensão na entrada positiva cair abaixo da tensão da entrada negativa, nesse caso, 0,1 VCC.

Inicialmente, o capacitor é carregado pelos dois resistores R_{ref} e R_x , e os pinos PD0 e PD3 são colocados como saída em nível alto. O capacitor se

⁴⁶ Ver os *applications notes* da Microchip: *Implementing Ohmmeter/Temperature Sensor e Resistance and Capacitance Meter Using a PIC 16C622 ou PEREIRA, Fábio. Microcontroladores MSP430 - Teoria e Prática*. 1 ed. São Paulo: Érica, 2005.

carrega pela resistência equivalente do paralelo entre R_{ref} e R_x ($R_{eq} = R_{ref} \parallel R_x$). Um tempo suficiente de carga é de aproximadamente $5R_{eq}C$.

Após carregado, o capacitor será descarregado por R_{ref} . Antes disso, o valor de contagem do TC1 é armazenado. Em seguida, o pino PD3 é configurado como entrada (alta impedância) e o pino PD0 é colocado em zero para a descarga do capacitor. Quando a tensão do capacitor cai abaixo de 0,1 VCC, a saída do comparador provoca a captura do TC1. A diferença do valor dessa captura e o valor do TC1 no início da descarga é o tempo de descarga do capacitor (t_{ref}).

Novamente, o capacitor deve ser carregado por R_{ref} e R_x , como feito no início do processo. Após isso, é a vez de descarregar o capacitor por R_x mantendo o pino de R_{ref} como entrada. O processo de contagem do tempo se faz do mesmo modo anterior e, assim, tem-se o tempo de descarga do capacitor por R_x (t_x). Na fig. 18.4, é exemplificado o processo para $R_{ref} = 10 \text{ k}\Omega$, $R_x = 3,3 \text{ k}\Omega$, $C = 1 \mu\text{F}$ e uma tensão de alimentação do microcontrolador de 5 V (VCC).

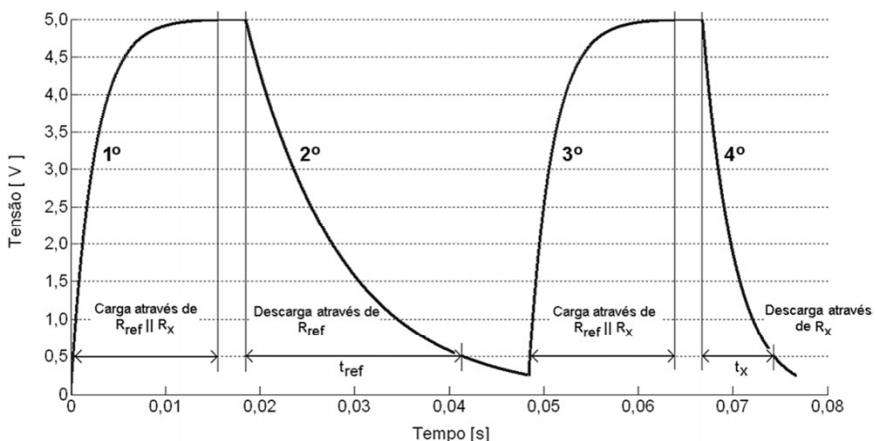


Fig. 18.4 – Carga do capacitor por $R_{ref} \parallel R_x$ e descarga por R_{ref} e R_x ($R_{ref} = 10 \text{ k}\Omega$, $R_x = 3,3 \text{ k}\Omega$ e $C = 1 \mu\text{F}$).

Como a tensão de descarga de um capacitor é dada por:

$$V_C = VCC \cdot e^{\left(\frac{-t}{RC}\right)} \quad [V] \quad (18.1)$$

e a tensão final de descarga de 0,1 VCC é a mesma para o R_X e R_{ref} , então:

$$VCC \cdot e^{\left(\frac{-t_{ref}}{R_{ref}C}\right)} = VCC \cdot e^{\left(\frac{-t_X}{R_X C}\right)} \quad (18.2)$$

o que resulta em:

$$R_X = R_{ref} \cdot \frac{t_X}{t_{ref}} \quad [\Omega] \quad (18.3)$$

Para uma medição precisa, os componentes utilizados no circuito devem ser de precisão e a tensão de referência deve ser provida com exatidão. A medição de resistência com o método acima pode ser empregada, por exemplo, para a leitura do valor da resistência de sensores.

CAPACITÂNCIA

Para o cálculo de uma capacidade desconhecida, o circuito da fig. 18.3 continua sendo empregado, com mudanças para avaliar o tempo de carga do capacitor desconhecido. Agora, o comparador analógico deve disparar o modo de captura do TC1 quando o valor da entrada positiva ultrapassar em mais de 0,63VCC o valor da entrada negativa⁴⁷. O circuito para análise é ilustrado na fig. 18.5.

⁴⁷ O objetivo desta seção é apresentar o conceito que pode ser empregado para o cálculo do valor de um capacitor. Os valores empregados no circuito podem ser diferentes de acordo com o programa utilizado. No caso apresentado, a tensão de 0,63 V deve ser provida por um circuito adequado.

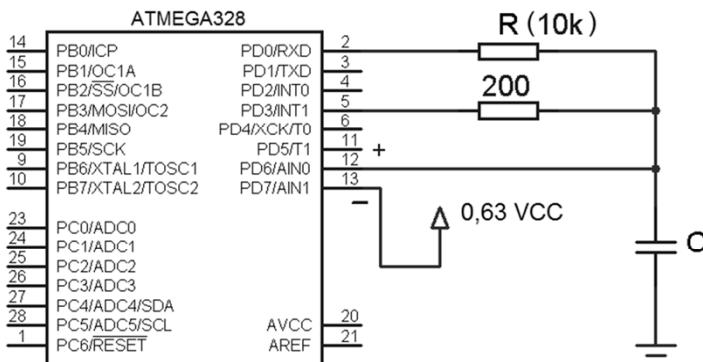


Fig. 18.5 – Medindo uma capacitância.

A tensão de carga de um capacitor é dada por:

$$V_C = VCC \left[1 - e^{\left(\frac{-t}{RC} \right)} \right] \quad [V] \quad (18.4)$$

Fazendo-se $t = RC$, resulta:

$$V_C = VCC \left[1 - e^{-1} \right] \cong 0,63 VCC \quad (18.5)$$

Assim, para se obter o valor de C basta saber o tempo de carga do capacitor até disparar o comparador:

$$C = \frac{t_{carga}}{R} \quad [F] \quad (18.6)$$

Para o funcionamento do método, primeiro o capacitor deve ser descarregado pelos dois resistores do circuito (pinos PD0 e PD3 como saídas em zero). Aqui deve ser estimado um tempo de descarga, limitando o valor máximo de capacitância que pode ser medida. Após a descarga do capacitor, o pino PD0 deve ser colocado em nível alto e o pino PD3 como entrada, para que o capacitor se carregue somente pelo resistor de 10 kΩ (R). O tempo de carga é computado entre o instante que se começa a carga do capacitor e o instante que o comparador analógico dispara o evento de captura do TC1.

18.2 LENDO MÚLTIPLAS TECLAS⁴⁸

Como o comparador analógico permite a captura do valor do TC1 em algum evento de comparação, a ideia de medida do tempo de carga de um capacitor pode ser empregada para a leitura de vários botões. Na fig. 18.6, é apresentado o circuito necessário para a leitura de N teclas. O procedimento adotado deve seguir os passos:

1. Configurar o pino AIN0 como saída e colocá-lo em nível lógico zero para descarregar o capacitor através do resistor de $220\ \Omega$.
2. Configurar o pino AIN0 como entrada do comparador analógico, tendo uma tensão de referência no pino AIN1 para o cômputo do tempo de carga do capacitor C1.
3. Usar o TC1 para medir o tempo que o comparador leva para disparar. Se o tempo medido for o maior possível (quando todos os resistores, R1 até RN participam da carga do capacitor), repete-se o processo. Caso contrário, com base no tempo calculado, determina-se qual botão foi pressionado.

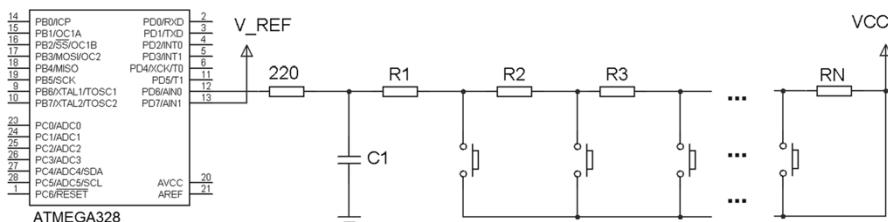


Fig. 18.6 – Lendo várias teclas com o comparador analógico baseado no tempo de carga de um capacitor.

O projetista deve calcular os valores dos resistores e capacitor empregados de acordo com o maior tempo de carga do capacitor, de tal forma que o tempo necessário para a identificação do pressionar de uma tecla não seja perceptível ao usuário.

⁴⁸ Application note da Microchip: *Tips 'n Tricks 8 pin PIC Microcontrollers*.

Exercícios:

18.1 – Elaborar um programa para executar uma determinada função quando uma tecla for pressionada. Use o comparador analógico para gerar a interrupção para o botão.

18.2 – Elaborar um programa para medir uma resistência desconhecida empregando o circuito da fig. 18.3.

18.3 – Como os circuitos apresentados para a medição de resistência e capacidade podem ser utilizados para se medir indutância? Quais as técnicas que podem ser empregadas para aumentar a resolução desses circuitos e como melhorá-los?

19. CONVERSOR ANALÓGICO-DIGITAL - ADC

Para o controle de variáveis externas por um sistema digital é necessária a interpretação de grandezas analógicas. Para tal, o emprego de conversores analógico-digitais torna-se imprescindível. Atualmente, os microcontroladores incorporam esses tipos de conversores, o que torna fácil o projeto de sistemas que necessitam ler variáveis analógicas. Neste capítulo, o conversor analógico-digital do ATmega328 é apresentado, juntamente com uma técnica para a leitura de um teclado matricial, o sensor de temperatura LM35, a sobreamostragem para aumentar a resolução do ADC e o filtro de média móvel para filtragem do sinal convertido.

O conversor AD do ATmega328 emprega o processo de aproximações sucessivas para converter um sinal analógico em digital. Suas principais características são:

- 10 bits de resolução (1024 pontos).
- Precisão de ± 2 LSBs (bits menos significativos).
- Tempo de conversão de 13 até 260 μ s.
- Até 76,9 kSPS (*kilo Samples Per Second*), 15 kSPS na resolução máxima.
- 6 canais de entrada multiplexados (+2 nos encapsulamentos TQFP e QFN/MLF).
- Faixa de tensão de entrada de 0 até VCC.
- Tensão de referência selecionável de 1,1 V.
- Modo de conversão simples ou contínua.
- Interrupção ao término da conversão.
- Eliminador de ruído para o modo *Sleep*.
- Sensor interno de temperatura com ± 10 °C de precisão.

O valor mínimo representado digitalmente é 0 V (GND) e o valor máximo corresponde à tensão do pino AREF menos 1 LSB; opcionalmente, a tensão do pino AVCC ou a tensão interna de referência de 1,1 V. O diagrama em blocos do conversor AD é ilustrado na fig. 19.1.

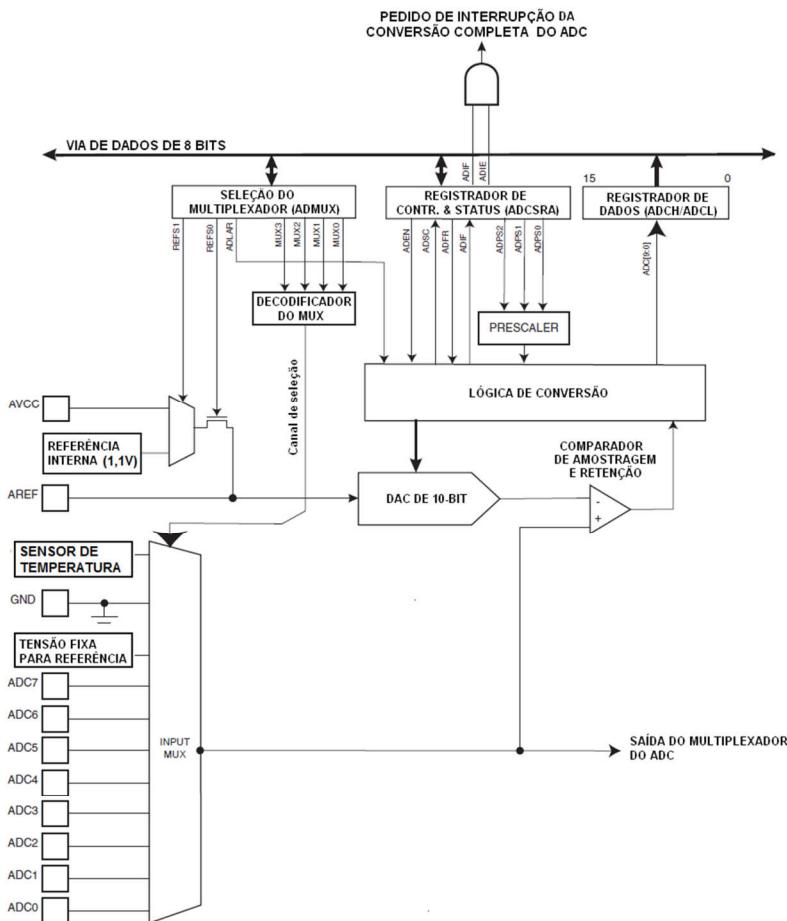


Fig. 19.1 – Diagrama do conversor AD do ATmega328.

O ADC produz um resultado de 10 bits que é escrito no registrador de 16 bits ADC (ADCH e ADCL). Por definição, o resultado é apresentado com ajuste à direita (ADCL tem os 8 LSBs). Opcionalmente, o resultado pode ter

ajuste à esquerda (ADCH com os 8 MSBs e os bits 7:6 do ADCL com os LSBs). Isso é feito, ajustando-se o bit ADLAR no registrador ADMUX. O ajuste à esquerda é interessante quando se deseja apenas 8 bits de resolução do ADC, bastando, nesse caso, apenas ler ADCH. Caso contrário, ADCL deve ser lido primeiro para garantir que o conteúdo do registrador pertence à mesma conversão. Quando a leitura do ADCL é realizada, os registradores de resultado são bloqueados para a atualização até que o ADCH seja lido. Nesse caso, o conteúdo de uma nova conversão não será salvo e a interrupção irá ocorrer mesmo com os registradores bloqueados.

Uma conversão simples é iniciada quando o bit ADSC é colocado em 1 no registrador ADCSRA. Esse bit se mantém em 1 durante a conversão e é limpo pelo hardware ao final desta. Se um canal diferente é escolhido para conversão, o ADC irá terminar a conversão corrente antes de mudar de canal. O resultado para uma conversão é dado por:

$$ADC = \frac{V_{IN} \cdot 1024}{V_{REF}} \quad (19.1)$$

onde V_{IN} é a tensão para conversão na entrada do pino e V_{REF} é a tensão selecionada de referência. O valor 0x000 representa a tensão zero e o valor 0x3FF representa V_{REF} menos 1 LSB.

Exemplo:

Se for empregada a tensão interna de referência de 1,1 V e a tensão de entrada for de 230 mV, qual o valor convertido pelo ADC? Qual o degrau de resolução do ADC (1 LSB)?

De acordo com a eq. 19.1, resultará:

$$ADC = \frac{0,23 \cdot 1024}{1,1} = 214 \quad \text{ou} \quad 0b11010110$$

A resolução do ADC será:

$$1LSB = \frac{V_{REF}}{1024} = \frac{1,1}{1024} = 1,07 \text{ mV}$$

No modo de conversão contínua (*Free Running*), o ADC é constantemente amostrado e os registradores de dados atualizados (bits ADTS2:0 = 0 no ADCSRB). A conversão começa escrevendo-se 1 no bit ADSC.

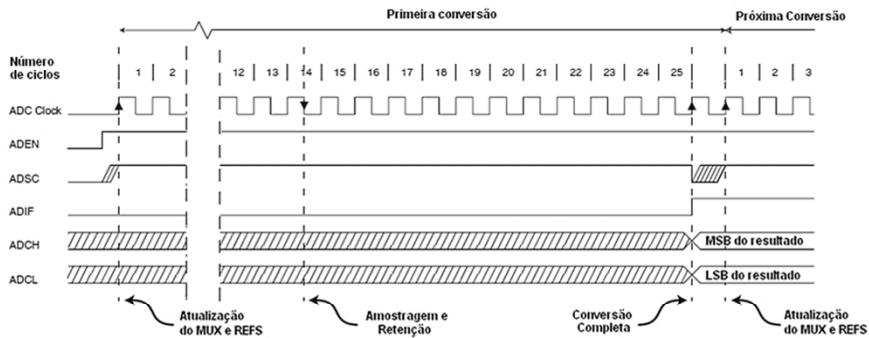
O circuito de aproximações sucessivas do ADC requer uma frequência de entrada entre 50 kHz e 200 kHz para obter a resolução máxima. Se uma resolução menor que 10 bits for desejada, uma frequência maior que 200 kHz pode ser empregada para se obter uma maior taxa de amostragem. Sinais de entrada com frequências maiores que a frequência de Nyquist ($f_{AD}/2$ - metade da frequência de amostragem) gerarão erros (*aliasing*). Eles devem ser previamente filtrados por um filtro passa baixa para eliminar o conteúdo de frequência acima da capacidade do ADC, para somente depois serem amostrados.

O ADC contém um módulo com *prescaler*, que aceita qualquer frequência de CPU acima de 100 kHz. O *prescaler* é ajustado nos bits ADPS do ADCSRA e começa a atuar quando o ADC é habilitado pelo bit ADEN. Enquanto esse bit for 1, o *prescaler* estará ativo.

Quando uma conversão simples é solicitada, colocando-se o bit ADSC em 1, a conversão inicia na próxima borda de subida do *clock* do ADC. Uma conversão normal leva 13 ciclos de *clock* do ADC. A primeira conversão leva 25 ciclos devido a inicialização do circuito analógico. A amostragem e retenção (*sample-and-hold*) leva 1,5 ciclos de *clock* após o início de uma conversão normal e 13,5 ciclos após o início da primeira conversão. Quando o resultado da conversão estiver completo, o resultado é escrito nos registradores de resultado, o bit ADIF é colocado em 1 e o ADSC é limpo. Os diagramas de tempo para a conversão simples são apresentados na fig. 19.2.

Quando uma conversão contínua estiver habilitada, uma nova conversão é iniciada logo após a anterior se completar, enquanto ADSC=1. O diagrama de tempo da conversão contínua é apresentado na fig. 19.3.

PRIMEIRA CONVERSÃO (MODO DE CONVERSÃO SIMPLES)



CONVERSÃO SIMPLES

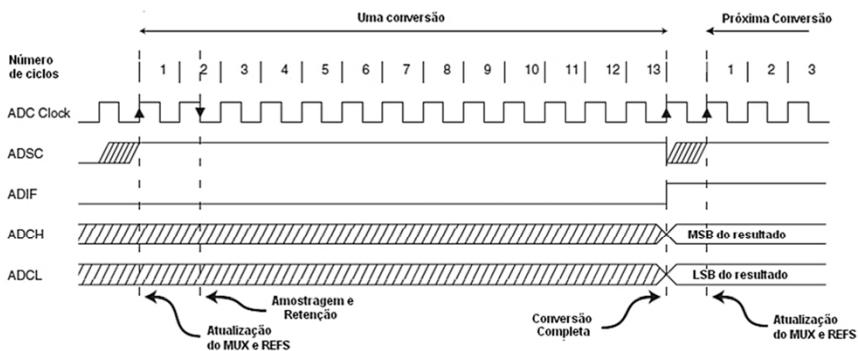


Fig. 19.2 – Diagramas de tempo para a conversão simples.

CONVERSÃO CONTÍNUA

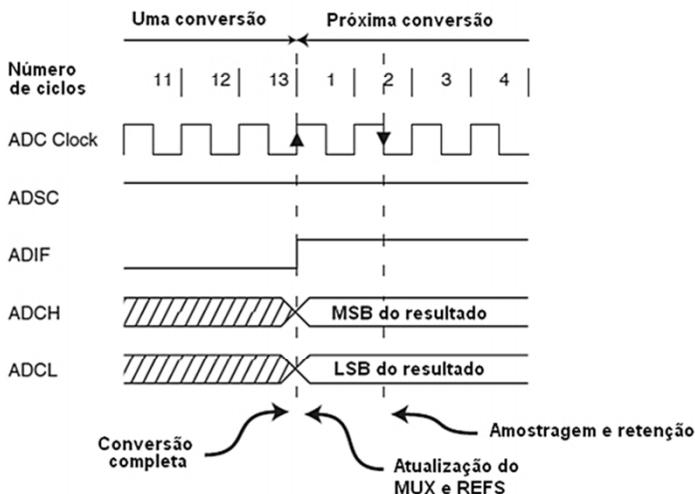


Fig. 19. 3 – Diagrama de tempo para a conversão contínua.

Cuidados devem ser tomados quando se muda o canal para a conversão. O registrador ADMUX responsável pela mudança pode ser atualizado seguramente nas seguintes ocasiões:

1. Quando o bit ADATE ou o bit ADEN for igual a zero.
2. Durante a conversão, pelos menos um ciclo de *clock* do ADC após o início da conversão.
3. Após a conversão, antes do *flag* de interrupção utilizado como disparo ser limpo.

Quando atualizado nessas condições, o ADMUX irá afetar a próxima conversão do ADC.

No modo de conversão simples, o canal deve sempre ser selecionado antes do inicio da conversão. No modo de conversão contínua, o canal deve ser selecionado antes da primeira conversão.

A referência de tensão para o ADC (V_{REF}) indica a faixa de conversão e pode ser: AVCC, referência interna de 1,1 V ou referência externa (no pino

AREF). Se a referência externa for utilizada, é recomendado o emprego de um capacitor entre o pino de AREF e o terra. Independente disso, a alimentação do ADC é feita pelo pino AVCC que não pode diferir de $\pm 0,3$ V de VCC. O circuito recomendado pela Atmel para a alimentação do ADC é apresentado na fig. 19.4 (filtro passa-baixa). Mesmo que o ADC não seja empregado, o pino AVCC deve ser ligado ao VCC.

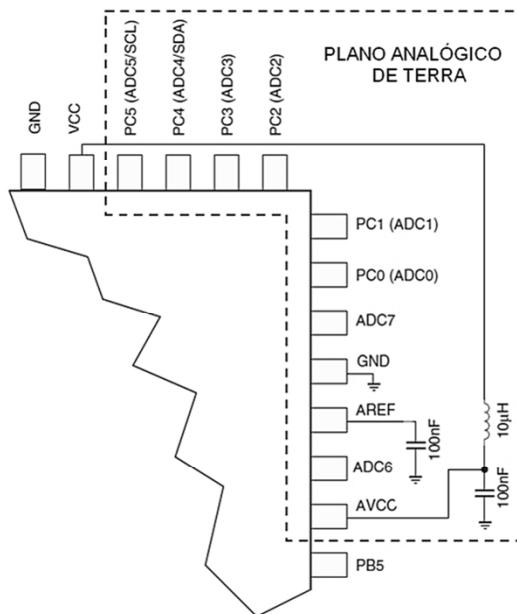


Fig. 19.4 – Circuito de alimentação recomendado para o ADC (encapsulamento TQFP).

O ADC possui um circuito eliminador de ruído que permite a conversão durante o modo *sleep*, visando reduzir o ruído induzido pela CPU e pelos periféricos de I/O. O eliminador de ruído pode ser usado no modo de redução de ruído do ADC ou no modo *Idle*. Para empregar essa característica, os seguintes passos devem ser observados:

1. O ADC deve estar habilitado e não estar realizando uma conversão. O modo de conversão simples deve estar selecionado e a interrupção de conversão completa habilitada.

2. Deve-se entrar no modo de redução de ruído do ADC ou modo *idle*. O ADC começará a conversão após a CPU parar.

Se nenhuma outra interrupção ocorrer antes da conversão do ADC estar completa, a interrupção do ADC irá despertar a CPU e executar a rotina de interrupção para a conversão completa do ADC. Se outra interrupção despertar a CPU antes do ADC completar a conversão, a interrupção será executada e o ADC só gerará a interrupção ao final de sua conversão. A CPU se manterá ativa até que um novo comando de *sleep* seja executado.

O ADC não será desligado automaticamente no modo *idle* ou de redução de ruído do ADC. Para evitar consumo de energia, o ADC deve ser desabilitado antes de se entrar num desses modos.

19.1 REGISTRADORES DO ADC

ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0
ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0
Lê/Escrive	L/E	L/E	L/E	L	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bits 7:6 – REFS1:0 – Reference Selection Bit

Estes bits selecionam a fonte de tensão para o ADC, conforme tab. 19.1. Se uma mudança ocorrer durante uma conversão, a mesma não terá efeito até a conversão ser completada (ADIF no ADCSRA estar ativo). A referência interna não pode ser utilizada se um tensão externa estiver sendo aplicada ao pino AREF.

Tab. 19.1 – Bits para a seleção da tensão de referência do ADC.

REFS1	REFS0	Seleção da Tensão de Referência
0	0	AREF, tensão interna V _{REF} desligada.
0	1	AVCC. Deve-se empregar um capacitor de 100 nF entre o pino AREF e o GND.
1	0	Reservado.
1	1	Tensão interna de referência de 1,1 V. Deve-se empregar um capacitor de 100 nF entre o pino AREF e o GND.

Bit 5 – ADLAR – ADC Left Adjust Result

Afeta a representação do resultado da conversão dos registradores de dados do ADC. ADLAR = 1, alinhado à esquerda; ADLAR = 0, alinhado à direita (ver a fig. 19.5). A alteração deste bit afeta imediatamente os registradores de dados.

Bits 3:0 – MUX3:0 – Analog Channel Selection Bits

Selecionam qual entrada analógica será conectada ao ADC (tab. 19.2).

Tab. 19.2 – Seleção do canal de entrada.

MUX3..0	Entrada
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	Sensor interno de temperatura
1001-1101	reservado
1110	1,1 V (tensão fixa para referência)
1111	0 V (GND)

ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0
ADCsra	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit 7 – ADEN – ADC Enable

Habilita o ADC. Se ADEN=0, o ADC é desligado. Desligar o ADC durante uma conversão irá finalizá-la.

Bit 6 – ADSC – ADC Start Conversion

No modo de conversão simples, ADSC=1 irá iniciar uma conversão. No modo de conversão contínuo, ADSC=1 irá iniciar a primeira conversão. ADSC ficará em 1 durante todo o processo de conversão e será zerado automaticamente ao seu término. Se o ADSC é escrito ao mesmo tempo em que o ADC é habilitado, a primeira conversão levará 25 ciclos de clock do ADC ao invés dos 13 normais.

Bit 5 – ADATE – ADC Auto Trigger Enable

Ativa o modo de auto disparo. O ADC começará uma conversão na borda positiva do sinal selecionado de disparo. A fonte de disparo é selecionada nos bits ADTS2:0 do registrador ADCSRB.

Bit 4 – ADIF – ADC Interrupt Flag

Este bit é ativo quando uma conversão for completada e o registrador de dados atualizado.

Bit 3 – ADIE – ADC Interrupt Enable

Este bit habilita a interrupção do ADC após a conversão se o bit I do SREG estiver habilitado.

Bits 2:0 – ADPS2:0 – ADC Prescaler Select Bits

Determinam a divisão do *clock* da CPU para o *clock* do ADC, conforme a tab. 19.3.

Tab. 19.3 – Seleção da divisão de *clock* para o ADC.

ADPS2	ADPS1	ADPS0	Fator de Divisão
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Exemplo do uso do prescaler do ADC:

É importante respeitar a máxima frequência de amostragem para o ADC, caso contrário a leitura apresentará erros. Assim, supondo o ADC trabalhando na sua máxima resolução (10 bits), a taxa máxima de trabalho é de 15 kSPS e, sabendo-se que uma conversão leva 13 ciclos de *clock* do AD, pode-se, então, determinar a máxima frequência desse *clock*.

Supondo o ATmega trabalhando a 16 MHz, resulta:

$$max_clk_{AD} = 15k \times 13 = 195 \text{ kHz}$$

Resultando num prescaler de:

$$prescaler = \frac{F_{CPU}}{max_clk_{AD}} = \frac{16 \text{ MHz}}{195 \text{ kHz}} = 82$$

Como não existe um prescaler com esse valor, escolhe-se 128, que permite respeitar o máximo de 15 kSPS, o que resultará em 9,6 kSPS.

Para diminuir possíveis erros devido a ruídos na conversão, quando não existe necessidade de se trabalhar na máxima frequência, é aconselhado utilizar a menor frequência possível para o ADC.

ADCL/ADCH – ADC Data Register

Quando a conversão estiver completa, o resultado é encontrado nesses dois registradores. Quando ADCL é lido, o registrador de dados não é atualizado até que o ADCH seja lido também. Consequentemente, se o resultado é alinhado à esquerda e não se necessitam mais que 8 bits de resolução, basta ler somente ADCH. Caso contrário, ADCL deve ser lido primeiro. Na fig. 19.5, é apresentado o alinhamento do resultado da conversão à direita e à esquerda, bit ADLAR=0 e ADLAR=1, respectivamente (do registrador ADMUX).

		15	14	13	12	11	10	9	8	
ADLAR=0		ADCH	-	-	-	-	-	ADC9	ADC8	
		ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
Bit		7	6	5	4	3	2	1	0	
Lê/Escrive		L	L	L	L	L	L	L	L	
		L	L	L	L	L	L	L	L	
Valor Inicial		0	0	0	0	0	0	0	0	
		0	0	0	0	0	0	0	0	
		15	14	13	12	11	10	9	8	
ADLAR=1		ADCH	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
		ADCL	ADC1	ADC0	-	-	-	-	-	-
Bit		7	6	5	4	3	2	1	0	
Lê/Escrive		L	L	L	L	L	L	L	L	
		L	L	L	L	L	L	L	L	
Valor Inicial		0	0	0	0	0	0	0	0	
		0	0	0	0	0	0	0	0	

Fig. 19.5 – Justificação do resultado à direita e à esquerda nos registradores de resultado do ADC.

ADCSR – ADC Control and Status Register B

Bit	7	6	5	4	3	2	1	0
ADCSR								
Lê/Escrive	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0
Valor Inicial	0	0	0	0	0	0	0	0

Bits 2:0 – ADTS2:0 - ADC Auto Trigger Source

Se o bit ADATE no registrador ADCSRA for 1, o valor desses bits seleciona a fonte para o disparo da conversão. Se ADATE é zero esses bits não têm efeito. As possíveis configurações para os bits ADTS2:0 são apresentadas na tab. 19.4.

Tab. 19.4 – Configurações para os bits ADTS2:0.

ADTS2	ADTS1	ADTS0	Fonte de disparo
0	0	0	conversão contínua
0	0	1	comparador Analógico
0	1	0	interrupção Externa 0
0	1	1	igualdade de comparação A do TC0
1	0	0	estouro de contagem do TC0
1	0	1	igualdade de comparação B do TC1
1	1	0	estouro de contagem do TC1
1	1	1	evento de captura do TC1

DIDR0 – Digital Input Disable Register 0

Bit	7	6	5	4	3	2	1	0
DIDR0	-	-	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E

Valor Inicial

Bits 5:0 – ADC5D:0D – ADC5:0 Digital Input Disable

Estes bits desabilitam individualmente as entradas digitais dos pinos do ADC. Deve(m) estar ativo(s) sempre que o pino correspondente for utilizado como entrada para o ADC, caso contrário deve(m) estar em zero.

19.2 MEDIÇÃO DE TEMPERATURA

O ATmega328 possui internamente um sensor de temperatura que é conectado a um canal do ADC. Ele é selecionado nos bits MUX3:0 do registrador ADMUX. Para o seu funcionamento é necessário selecionar a tensão interna de referência de 1,1 V. A relação entre a tensão gerada e a temperatura é de aproximadamente 1 mV/°C (mais ou menos 1 LSB/°C) e a precisão é de ± 10 °C. Alguns valores típicos são apresentados na tab. 19.5.

Tab. 19.5 – Temperatura versus tensão de saída do sensor interno do ATmega328.

Temperatura	- 45 °C	+ 25 °C	+ 85 °C
Tensão	242 mV	314 mV	380 mV

Devido ao processo de fabricação, o comportamento com a temperatura varia de chip para chip. Todavia, é possível obter resultados mais exatos, até ± 2 °C ou ainda melhores, realizando a calibração do sensor através do software de aplicação⁴⁹. Entretanto, isso não é conveniente quando se deseja obter resultados precisos, reproduzíveis e quando se quer simplicidade na programação.

Considerando-se 10 bits para a conversão do AD, a temperatura é determinada por:

$$Temp = \frac{(ADCL | (ADCH \ll 8)) - Offset}{k} \quad (19.2)$$

onde ADCL e ADCH são os registradores do ADC que possuem o valor da conversão, *Offset* é um erro fixo, ajustado manualmente, e *k* é um coeficiente fixo (também deve ser ajustado manualmente).

A seguir, é apresentado um programa para enviar a temperatura do ATmega para um computador utilizando a USART (ver o capítulo 15).

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL //define a frequência do microcontrolador - 16MHz
#include <avr/io.h> //definições do componente especificado
#include <util/delay.h> //biblioteca para o uso das rotinas de atraso
#include <avr/pgmspace.h> //para o uso do PROGMEM, gravação de dados na flash
#include <avr/interrupt.h> //para o uso de interrupções

//Definições de macros para o trabalho com bits
#define set_bit(y,bit) (y|=(1<<bit))
#define clr_bit(y,bit) (y&=~(1<<bit))
#define cpl_bit(y,bit) (y^=(1<<bit))
#define tst_bit(y,bit) (y&(1<<bit))

#endif
```

⁴⁹ Application note da Atmel: AVR122: *Calibration of the AVR's internal temperature reference.*

Sensor_temp_int_ATmega.c (programa principal)

```
//=====
// Lendo o sensor interno de temperatura do ATmega328          //
//=====
#include "def_principais.h"
#include "USART.h"

#define Offset_temp 363           //valor do offset de temperatura

unsigned char digitos[tam_vetor];/*declaração da variável para armazenagem dos dígitos
(tam_vetor está em USART.h)*/
const char msg[] PROGMEM = "Sensor Interno de Temperatura do ATmega328\n\0";

signed int le_temp();
//-----
int main(void)
{
    USART_Inic(MYUBRR);

    //Tensão interna de ref (1.1V), sensor de temp. do chip
    ADMUX = (1<<REFS1) | (1<<REFS0) | (1<<MUX3);

    //habilita o AD e define um prescaler de 128 (clk_AD = F_CPU/128), 125 kHz
    ADCSRA = (1<<ADEN) | (1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
    escreve_USART_Flash(msg);

    while(1)
    {
        ident_num((unsigned int)le_temp(),digitos); //leitura de temperatura sem sinal
        USART_Transmite(digitos[2]);
        USART_Transmite(digitos[1]);
        USART_Transmite(digitos[0]);
        USART_Transmite(176);           //símbolo 'o'
        USART_Transmite('C');
        USART_Transmite('\n');        //nova linha
        _delay_ms(1000);
    }
}
//-----
signed int le_temp()
{
    set_bit(ADCSRA, ADSC);      //inicia a conversão
    while(tst_bit(ADCSRA,ADSC)); //espera a conversão ser finalizada

    return (ADC - Offset_temp);   //fator k de divisão = 1
}
//-----
```

USART.h e **USART.c** vistos no capítulo 15.

Exercício:

19.1 – Com base no *application note* AVR122 faça um programa para ler com a maior precisão possível o sensor de temperatura interno do ATmega328. Utilize uma média de leituras para apresentar o resultado.

19.3 LENDO UM TECLADO MATRICIAL⁵⁰

Uma técnica interessante para ler um teclado no formato matricial é utilizar o ADC para converter um sinal de tensão proveniente de uma rede resistiva. Na fig. 19.5, é apresentado um teclado com 16 teclas com um arranjo adequado de resistores. Assim, quando uma determinada tecla for pressionada, será gerada uma tensão única sobre o resistor R9. De acordo com a tensão gerada, se determina qual tecla foi pressionada. Com nenhuma tecla pressionada, a tensão sobre o resistor R9 será nula.

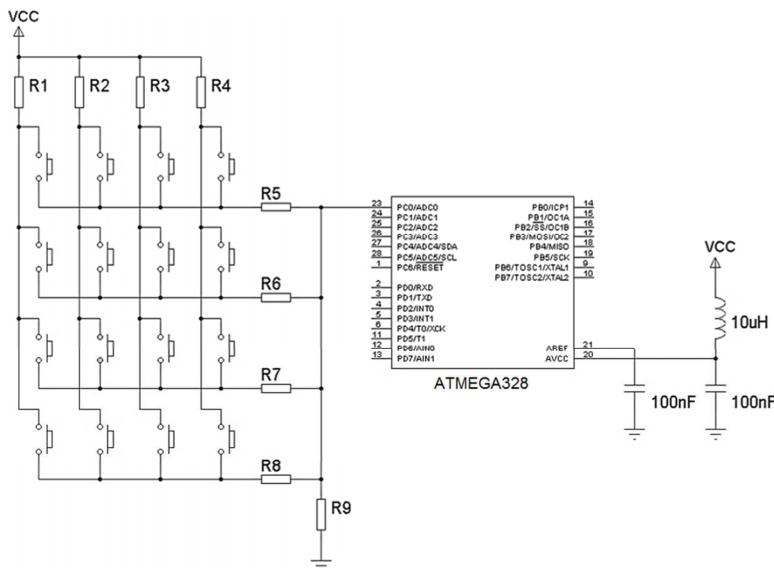


Fig. 19.5 – Leitura de um teclado 4×4 com o uso do ADC.

⁵⁰ Application note da Microchip: *Tips 'n Tricks 8 pin PIC Microcontrollers*.

Para que o sistema funcione adequadamente, é necessário calcular os resistores empregados garantindo 16 faixas de tensão diferentes para a entrada do ADC (fig. 19.5). Nesse caso, é importante empregar resistores de precisão para evitar variações consideráveis de tensão para cada tecla. Para a estabilidade da leitura, na avaliação dos dados do ADC, o programa deve considerar uma faixa de tensão pertinente para cada tecla. A tensão de referência para o ADC deve ser a tensão de alimentação do sistema.

Apesar de ser possível realizar leituras de várias teclas com o ADC, um número inferior a 16 teclas é mais fácil de ser implementado, pois não existirão valores de resistores muito próximos entre si. Assim, teclados matriciais com 12 teclas, 9 teclas ou menos poderão ser feitos com resistores comerciais facilmente encontrados.

Exercícios:

19.2 – Determine valores para os resistores da fig. 19.5. Depois repita o cálculo para um teclado matricial de 12 e 9 teclas.

19.3 – Empregando o ADC do ATmega328, elaborar um circuito e o respectivo programa para medir com precisão uma resistência desconhecida.

19.4 – Faça um programa para a leitura do teclado da fig. 19.6.

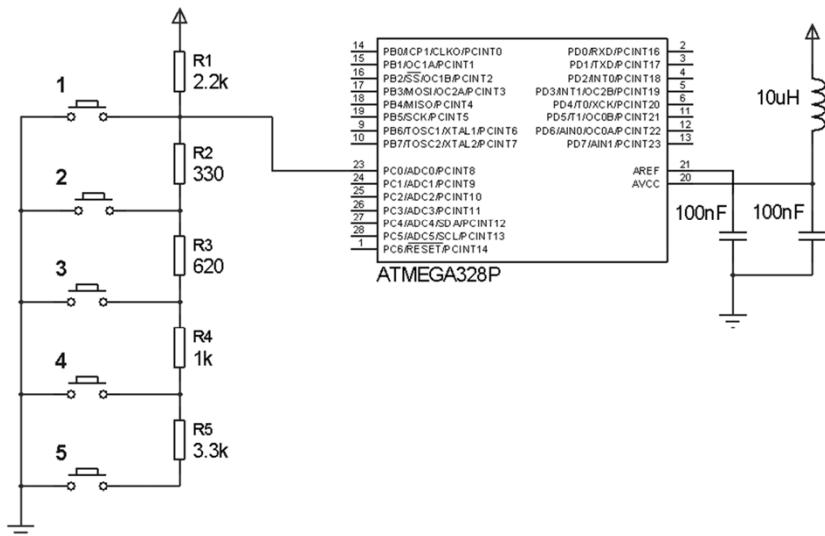


Fig. 19.6 – Leitura de 5 teclas com o ADC do ATmega328.

19.4 SENSOR DE TEMPERATURA LM35

O LM35 é um sensor de temperatura analógico de precisão de aproximadamente $0,25\text{ }^{\circ}\text{C}$ produzido pela National Semiconductors. Apresenta resposta linear de $10\text{ mV}/^{\circ}\text{C}$ (a $0\text{ }^{\circ}\text{C}$ deve fornecer 0 V de saída) e uma faixa de operação de $-55\text{ }^{\circ}\text{C}$ até $+150\text{ }^{\circ}\text{C}$ (dependendo do modelo). Está disponível em vários encapsulamentos, mas o mais comum é o T092 (similar ao dos transistores BC). As vantagens do LM35 são seu custo e a facilidade de integração com microcontroladores que dispõem de ADC.

Caso haja necessidade da leitura de temperaturas negativas, é necessário o emprego de uma fonte negativa de tensão em conjunto com um resistor, ou o uso de um diodo entre o pino de terra e o terra do circuito, com um resistor entre o pino de saída e o terra.

Por ser mais barato e responder a maioria dos requisitos de projeto, geralmente se emprega o LM35D, cuja faixa de temperatura é só positiva, se estendendo de 0 °C até + 100 °C.

Na fig. 19.7, é apresentado o circuito resumido para o teste do LM35 no Arduino⁵¹. A informação de temperatura é enviada a um computador (ver o capítulo 15). Na sequência, é apresentado o programa desenvolvido.

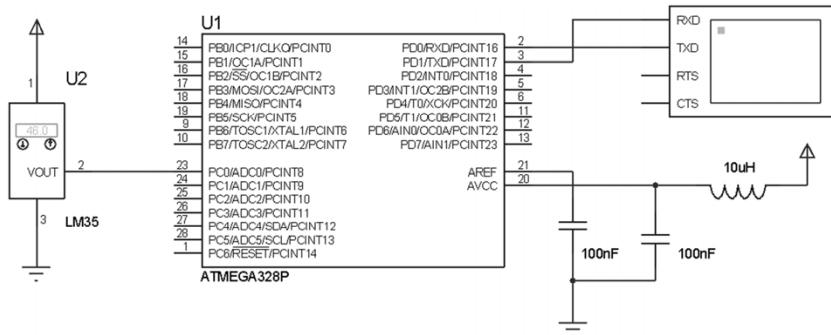


Fig. 19.7 – Analise de temperatura com o LM35 com envio de dados a um computador.

LM35.c (programa principal)

```
//=====
// Sensor de temperatura LM35 - envio de dados para o PC
// Resolução de 1 grau Centigrado
//=====
#include "def_principais.h"
#include "USART.h"

const char msg[] PROGMEM = "Sensor de Temperatura LM35\n\0";

unsigned int temp;
unsigned char digitos[tam_vetor];
//-----
int main()
{
    DDRC = 0x00;
    PORTC = 0xFE;
    USART_Inic(MYUBRR);
}
```

⁵¹ A plataforma Arduino não segue a recomendação da Atmel quanto ao uso do indutor ligado ao pino AVCC e do capacitor ligado ao pino AREF (ver circuito na fig. 3.2).

```

//configura ADC
ADMUX = 0b11000000; //Tensão interna de ref (1.1V), canal 0
ADCSRA = 0b11101111; /*habilita o AD, habilita interrupção, modo de conversão
contínua, prescaler = 128*/
ADCSRB = 0x00;      //modo de conversão contínua
set_bit(DIDR0,0);   //desabilita pino PC0 como I/O, entrada do ADC0

sei();

escreve_USART_Flash(msg);

while(1)
{
    ident_num(temp,digitos);
    USART_Transmite(digitos[3]);
    USART_Transmite(digitos[2]);
    USART_Transmite(digitos[1]);
    USART_Transmite(',');
    USART_Transmite(digitos[0]);
    USART_Transmite(176);//simbolo 'o'
    USART_Transmite('C');
    USART_Transmite('\n');
    _delay_ms(1000);
}
//-----
ISR(ADC_vect)
{
    temp = ADC + (ADC*19)/256;

    /* O LM35 apresenta uma saída de 10mV/°C. O valor de leitura do AD é dado por
    ADC = Vin*1024/Vref, como Vref = 1,1V, para converter o valor do AD para graus
    Celsius, é necessário multiplicar o valor ADC por 1100/1024 (considerando uma
    casa decimal antes da vírgula). Utilizando a simplificação matemática e mantendo
    a variável temp com 16 bits, resulta: 1100/1024 = 1 + 19/256 */
}

//-----

```

def_principais.h foi apresentado anteriormente e **USART.h** foi apresentado no capítulo 15.

Exercício:

19.5 – Elaborar um programa para ler os sensores de temperatura LM35 conforme o circuito da fig. 19.8. Escreva a temperatura de cada um deles em uma linha do LCD 16×2.

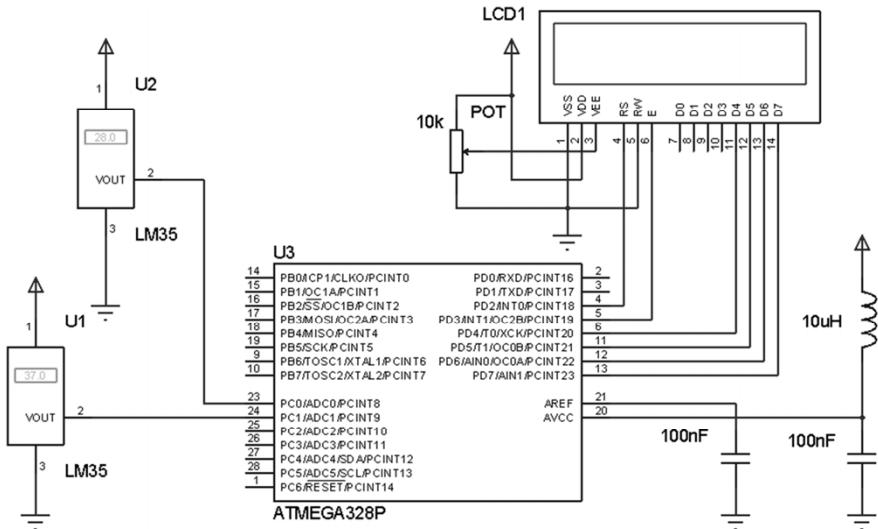


Fig. 19.8 – Empregando o ATmega328 para ler dois sensores de temperatura LM35.

19.5 SOBREAMOSTRAGEM

O ADC do ATmega possui uma resolução máxima de 10 bits, ou seja permite 1024 níveis de discretização. Entretanto, os dois bits menos significativos são imprecisos, o que faz com que muitos programadores utilizem somente os 8 bits mais significativos, resultando em 256 níveis de discretização. Nesse caso, é só habilitar o bit ADLAR no registrador ADMUX e ler somente o registrador ADCH.

É possível aumentar a resolução do ADC com uma técnica simples, chamada sobreamostragem. Dessa forma, a imprecisão da leitura pode ser diminuída. Essa técnica só pode ser empregada no modo de conversão contínua.

A cada diminuição de 4 vezes na frequência de amostragem do AD, a sua resolução pode ser aumentada em um bit⁵²:

$$\frac{f_{amostragem\,AD}}{4} \rightarrow (\text{ADC_soma} \gg 1)$$

onde ADC_soma é a soma de 4 valores consecutivos amostrados normalmente pelo ADC e “ $\gg 1$ ” é o deslocamento dessa soma um bit para a direita, o que elimina o seu bit menos significativo. Assim, para uma frequência de amostragem de 15 kHz, se essa taxa for diminuída para 3,75 kHz, a resolução do ADC do ATmega328 pode passar de 10 bits para 11 bits. Se a frequência for dividida de novo por 4 (divisão de 15 kHz pelo total de 16) a resolução pode passar para 12 bits e, assim por diante. O valor da conversão é somado durante o intervalo da amostragem para o efetivo aumento da resolução da conversão. Depois, o valor somado é deslocado um bit para a direita a cada diminuição de 4 vezes na frequência de amostragem. O processo é ilustrado na fig. 19.9.

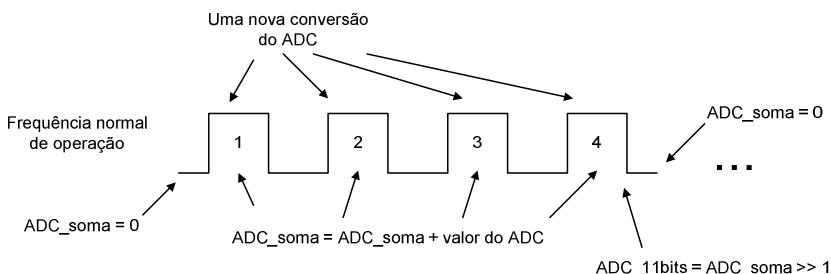


Fig. 19.9 – Esquema de sobreamostragem para aumentar a resolução do ADC de 10 para 11 bits.

Em resumo, a frequência normal de trabalho do ADC gera uma nova amostra a cada intervalo de conversão, representados pelos números 1 a 4 na fig. 19.9. A cada nova conversão, o valor do ADC deve ser somado em uma variável auxiliar, no caso ADC_soma. Após 4 conversões, a variável que contém a soma deve ser deslocada 1 bit para a direita (dividida por 2). Esse valor deslocado é o novo valor do ADC, agora com uma resolução de 11

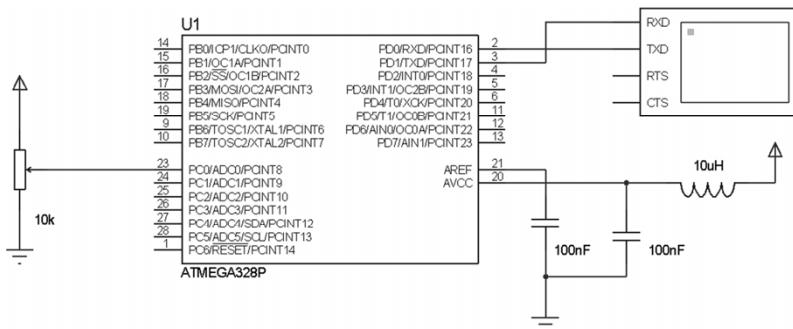
⁵² Application Note da Atmel: AVR121: *Enhancing ADC resolution by oversampling*.

bits. A frequência de amostragem é 4 vezes menor porque somente após quatro conversões normais do ADC é que é gerado um novo valor com 11 bits. A seguir, é apresentado um trecho de código que ilustra a aplicação da sobreamostragem para se obter o aumento de um bit na resolução do ADC.

```
//-----
// Sobreamostragem - aumentando a resolução do ADC de 10 para 11 bits
//-----
unsigned int ADC_11bits;
. . .
ISR(ADC_vect)
{
    static unsigned int ADC_soma=0;
    static unsigned char cont=4;
    ADC_soma += ADC;      //soma o valor do ADC na variável para o aumento da resolução
    cont--;
    if(cont==0)
    {
        ADC_11bits = ADC_soma >> 1;    //deslocamento de 1 bit para a direita
        ADC_soma=0;
        cont=4;
    }
}
//-----
```

Exercício:

19.6 – Elaborar um programa para ler a tensão proveniente de um potenciômetro com o ADC do ATmega328 (ver a fig. 19.10). Apresente o valor binário no computador (ver o capítulo 15). Depois aumente a resolução do ADC para 11 bits e 12 bits, respectivamente. Verifique os resultados.



Obs.: potenciômetros são comuns em dispositivos de controle de posição, como por exemplo, *joysticks*.

19.6 FILTRO DE MÉDIA MÓVEL

Muitas vezes, para eliminar ou diminuir algum ruído indesejável em um sinal, é necessário filtrá-lo. Isso é comum quando se trabalha com sinais provenientes do ADC. Essa filtragem é, então, feita através da programação. O problema é que a teoria sobre filtros digitais é complexa e o programador a evita sempre que possível. Todavia, pode-se utilizar um filtro bem simples, chamado de média móvel, que pode resolver o problema do ruído.

O filtro de média móvel é obtido calculando-se a média de um conjunto de valores, sempre se adicionando um novo valor ao conjunto e se descartando o mais velho. Não é apenas uma média de um conjunto isolado de valores. O filtro de média móvel é representado por:

$$y[n] = \frac{1}{N+1} \sum_{k=0}^N x[n-k] \quad (19.3)$$

onde n é o tempo atual (é o índice dos vetores utilizados), $N+1$ é o número de amostras utilizadas para a filtragem, $y[n]$ é o sinal filtrado e $x[n-k]$ representa o conjunto dos valores a serem somados. A eq. 19.3 pode ser representada pelo diagrama da fig. 19.11, com os valores de $b_0, b_1 \dots b_N$ iguais a $1/(N+1)$.

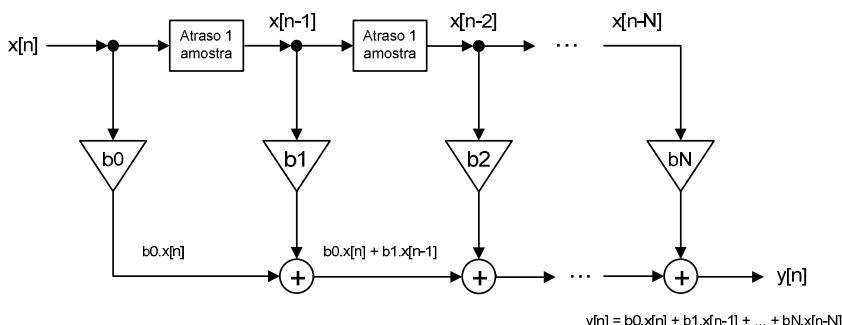


Fig. 19.11 – Diagrama de um filtro digital não recursivo.

O diagrama da fig. 19.11 representa um filtro chamado não recursivo, pois o sinal de saída $y[n]$ depende somente do sinal de entrada $x[n]$. Caso o sinal de saída dependesse de valores passados da saída, o filtro seria chamada recursivo. No projeto de um filtro digital, determinam-se os coeficientes de multiplicação para as amostras, no caso b_0, b_1, \dots, b_N para o diagrama citado. Esses coeficientes podem ser determinados com o uso de alguma ferramenta computacional como o MATLAB® ou MATCAD®, e o programador apenas necessita realizar as somas e multiplicações nas amostras certas. A qualidade do filtro e o tipo de filtro (passa baixa, passa alta, passa faixa) vai depender do número de coeficientes utilizados (quantidade de amostras) e dos seus valores.

Ao utilizar coeficientes fixos, o filtro de média móvel produz um filtro passa baixa suave, reduzindo os sinais de alta frequência. Caso se deseje outro tipo de filtragem, será necessário a multiplicação com valores fracionários, o que exigirá o uso de ponto flutuante no programa. Isso pode ser um problema para microcontroladores de 8 bits pelo consumo maior de memória e da limitada capacidade de processamento da CPU.

Na fig. 19.12, é apresentada a resposta em frequência de um filtro de média móvel para 16 amostras. O eixo horizontal é a frequência em Hertz, o número 1 representa a frequência de amostragem do sinal dividida por 2. Desta forma, supondo uma frequência de amostragem 20 kHz, cada linha vertical do gráfico corresponderia a 1 kHz.

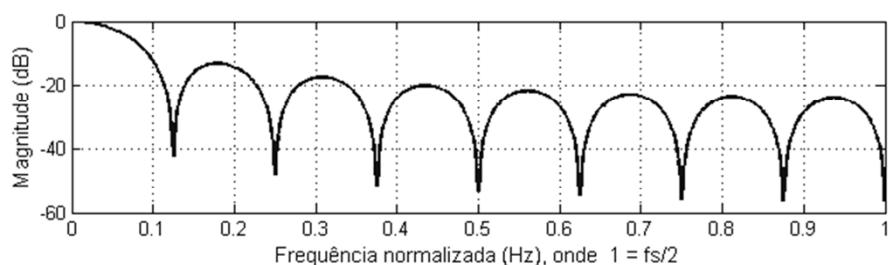


Fig. 19.12 – Resposta em frequência de um filtro de média móvel de 16 amostras.

A resposta apresentada na fig. 19.12 mostra que o desempenho de um filtro de média móvel é razoável, estando a atenuação dos sinais indesejados na faixa de aproximadamente -20 dB (90 %).

A seguir, é apresentado um exemplo para a programação de um filtro de média móvel de 16 amostras para os valores convertidos pelo ADC do ATmega.

```
-----  
// Filtro de média móvel com 16 amostras para o sinal do ADC  
-----  
unsigned int x[16], y;  
. . .  
ISR(ADC_vect)  
{  
    unsigned char i=15;  
  
    y=0;  
    do  
    {  
        x[i] = x[i-1]; //anda um passo nas amostras anteriores  
        y += x[i];  
        i--;  
  
    } while (i!=0);  
  
    x[0]=ADC;           //nova amostra entra no filtro  
    y = (y + x[0])/16; //sinal filtrado  
}  
-----
```

Exercício:

19.7 - Utilize um filtro de média móvel de 5 amostras para o exercício 19.6.

20. MESCLANDO PROGRAMAÇÃO C COM ASSEMBLY

Antigamente, os microcontroladores dispunham de pouca memória de programa e o uso dos compiladores C não era comum. Com o desenvolvimento da microeletrônica, a quantidade de memória de programa aumentou consideravelmente e o seu custo diminuiu, o que fez com que os compiladores C se multiplicassem.

Como programar em C é muito mais fácil e rápido, é difícil encontrar um programador que utilize rotinas em *assembly* ou exclusivamente o *assembly*. Entretanto, apesar dos compiladores C terem evoluído, maior eficiência e densidade de código só podem ser conseguidas com o código puramente em *assembly*.

Como programar em C sacrifica um pouco da memória de programa e do desempenho, em aplicações feitas em C, onde o tempo de execução e/ou a quantidade de memória são pontos críticos de projeto, caso a otimização em C não resolva o problema, pode ser necessário incluir rotinas em *assembly* para otimizar manualmente partes do programa.

O compilador GCC desenvolvido para os microcontroladores AVR de 8 bits da Atmel permite a elaboração de um projeto com códigos fontes escritos em C e *assembly* (extensão .S), desde que se defina o projeto como GCC. O projeto GCC desabilita o *assembly* nativo do programa AVR Studio e todas os códigos fontes passam a ser manipulados por esse *plug-in*.

Em um projeto GCC, os códigos fonte em C são compilados antes dos fontes em *assembly*, o que traz as seguintes implicações:

- As funções em C chamadas a partir do *assembly* não precisam ser declaradas na fonte *assembly*, uma vez que elas já se encontram na tabela de símbolos.
- As variáveis globais devem ser declaradas nos arquivos C, onde elas são inicializadas com o valor zero.

- As funções *assembly* chamadas pelo C devem ser declaradas como externas nos arquivos C e globais nos arquivos *assembly*.
- Tratadores de interrupções em *assembly* devem ser declarados como globais nos arquivos *assembly*.

O processo de criação do arquivo compilado *.hex para a gravação do microcontrolador com base nos arquivos C e *assembly* é apresentado na fig. 20.1⁵³.

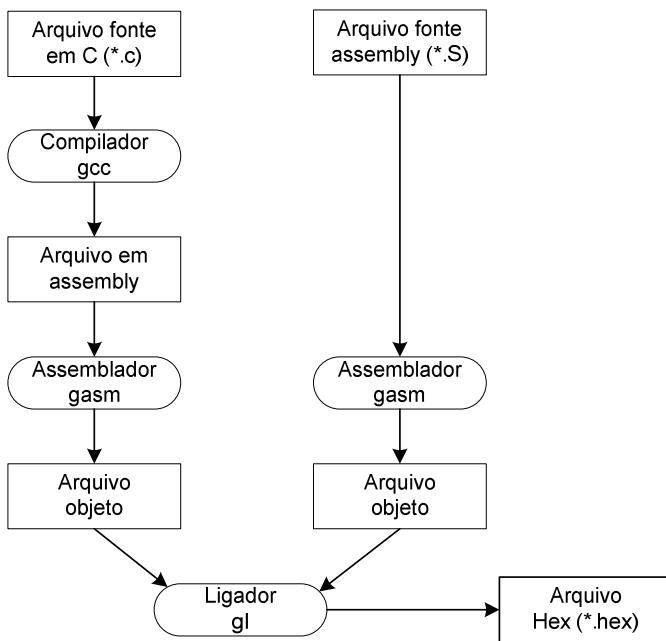


Fig. 20.1 – Processo de criação do arquivo *.hex a partir de arquivos fontes em C e assembly.

⁵³ O termo *Assemblador* não existe na língua portuguesa, a melhor tradução seria: montador assembly (conversor do arquivo fonte assembly para o código de máquina). O termo ligador corresponde a *linker* em inglês, ele faz a conexão entre os arquivos objeto, previamente compilados, para um único arquivo *.hex.

20.1 USO DOS REGISTRADORES DO AVR

Para a programação em *assembly*, o GCC utiliza determinados conjuntos dos 32 registradores de uso geral do AVR, distribuídos da seguinte forma:

- R2–R17, R28, R29 – estes registradores podem ser utilizados pelo GCC para armazenar dados locais. As sub-rotinas em C os deixam inalterados. Se uma sub-rotina em *assembly* utilizar qualquer um desses registradores, ela será responsável por salvá-los na pilha e restaurá-los antes do retorno.
- R18–R27, R30, R31 – estes registradores podem ser utilizados pelo GCC para armazenar dados locais. Eles podem ser utilizados livremente em sub-rotinas *assembly*. Sub-rotinas em C podem se apropriar de alguns deles, assim a rotina *assembly* que as chama é responsável por preservá-los.
- R0 – registrador temporário. Se utilizado pelo código *assembly*, ele deve ser salvo antes da chamada de um código C, pois esse pode modificá-lo. Os tratadores de interrupção são uma exceção, uma vez que preservam o registrador R0.
- R1 – o compilador C assume que este registrador contém zero. Se for utilizado no *assembly*, deve ser limpo após o uso (CLR R1). Tratadores de interrupção salvam e limpam R1 na entrada, e o restauram na saída (no caso dele não ser zero).

20.2 CONVENÇÃO NA CHAMADA DE FUNÇÕES

Nas chamadas de funções, os argumentos são alocados da esquerda para a direita, de R25 a R8, e são alinhados para iniciar em registradores de numeração par. Todos os argumentos ocupam um número par de registradores. Argumentos para funções com lista de argumentos variável (printf, scanf, etc.) são passados pela pilha e argumentos **char** são estendidos para **int**.

Valores retornados: 1 byte em R24, 2 bytes em R25-R24, 4 bytes em R25-R22 e 8 bytes em R25-R18. Valores retornados de 1 byte são estendidos para 16 bits pelo C, sendo variáveis **unsigned char** mais eficientes que **signed char**, uma vez que para utilizar as primeiras basta limpar R25.

20.3 EXEMPLOS

A seguir serão apresentados alguns exemplos para elucidar como mesclar C e *assembly* no AVR Studio.

Programa em *assembly* chama funções em C

Neste exemplo, um programa em *assembly* chama funções escritas em C. O programa escreve na linha superior de um LCD 16×2 a mensagem "Usando ASM com C" (o barramento de dados de 4 bits foi empregado com instruções simplificadas, ver o capítulo 5). Para a criação dos arquivos no AVR Studio, a seguinte sequência deve ser seguida:

1. Criar um projeto em C e o arquivo C que conterá as funções chamadas pelo *assembly*. O nome do arquivo deve ser o mesmo do projeto.
2. Criar o arquivo *assembly* que conterá o código principal, salvar com extensão .S e anexar ao projeto⁵⁴. O nome deve ser diferente do arquivo C.

⁵⁴ Obs.: na versão utilizada do AVR Studio 5.1, o arquivo *.S aparece com o texto totalmente em preto, sem a distinção de cores entre os termos da programação. Isto é uma falha do AVR Studio, mas não interfere no resultado.

Prog_Princ.S (Programa principal, vai chamar as funções em C)

```
/*
-----O assemblador GCC usa o mesmo pré-processador que o GCC C/C++ compiler.
Assim ele usa of #include em vez de .include.*/
#include <avr/io.h>

/* O define a seguir é necessário para subtrair 0x20 dos endereços de I/O. */

#define __SFR_OFFSET 0

/* O assemblador GCC permite a inicialização de dados no segmento de dados.
Os dado estão realmente armazenados na memória de programa. O assemblador
gera um código de partida que copia os dados inicializados na SRAM. */

.section .data
msg: .ascii "Usando ASM com C"

/* Aqui inicia o segmento de código */
.section .text

.global main
main:    ldi      r16,0xff
          out     DDRD,r16
          rcall   inic_LCD      //esta função não pede argumentos
          ldi      r25, hi8(msg)
          ldi      r24, lo8(msg)  //ponteiro de 16 bits passado por r25:r24
          rcall   escreve_LCD
halt:    rjmp   halt
          .end
//-----
```

Assembly_chama_C.c (arquivo com as funções em C)

```
/*
-----#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

#define set_bit(adress,bit) (adress|=(1<<bit))
#define clr_bit(adress,bit) (adress&=~(1<<bit))
#define tst_bit(adress,bit) (adress&(1<<bit))
#define cpl_bit(adress,bit) (adress^=(1<<bit))

#define DADOS_LCD PORTD
#define RS      PD2
#define E       PD3
//-----
//Sub-rotina para enviar comandos ao LCD com dados de 4 bits
//-----
void cmd_LCD(unsigned char c, char cd)
{
    static unsigned char i=1;

    DADOS_LCD = c;//primeiro os 4 MSB. (PD4 - PD5 - PD6 - PD7) -> (D4 - D5 - D6 - D7 LCD)

    for(;i!=0;i--)
    {
        if(cd==0)
            clr_bit(PORTD,RS);
```

```

    else
        set_bit(PORTD,RS);

    set_bit(PORTD,E);
    clr_bit(PORTD,E);
    _delay_us(45);

    if((cd==0) && (c<4))//se for instrução espera o tempo de resposta do LCD
        _delay_ms(2);

    DADOS_LCD = c<<4;
}
i = 2;
}

//-----
//Sub-rotina de inicialização simplificada para 4 bits de dados do LCD
//-----
void inic_LCD(void) //envio de instruções para o LCD
{
    cmd_LCD(0x28,0); //interface de 4 bits
    cmd_LCD(0x28,0); //interface de 4 bits 2 linhas (aqui se habilita as 2 linhas)
    cmd_LCD(0x0c,0); //mensagem aparente cursor inativo não piscando
    cmd_LCD(0x01,0); //limpa todo o display
    cmd_LCD(0x80,0); //desloca cursor para a posição 0x80
}
//-----
//Sub-rotina de escrita no LCD
//-----
void escreve_LCD(char *c)
{
    for (; *c!=0;c++) cmd_LCD(*c,1);
}
//-----

```

Programa em C chama função em assembly

Neste exemplo, um programa em C chama uma função escrita em *assembly*. O programa escreve na linha superior do LCD a mensagem "Usando C com ASM" (o barramento de dados de 4 bits foi empregado com instruções simplificadas, ver o capítulo 5). Para a criação dos arquivos no AVR Studio, a seguinte sequência deve ser seguida:

1. Criar um projeto em C e o arquivo C que irá chamar a função em *assembly*. O nome do arquivo deve ser o mesmo do projeto.
2. Criar o arquivo *assembly* que conterá a função, salvar com extensão .S e anexar ao projeto. O nome deve ser diferente do arquivo C.

C_chama_Assembly.c (programa principal)

```
-----  
#define F_CPU 16000000UL  
#include <avr/io.h>  
#include <util/delay.h>  
#include <avr/pgmspace.h>  
  
unsigned char msg=1;//usado no assembly  
const unsigned char msg1[] PROGMEM = "C COM ROTINA ASM\0";//mensagem na memória flash  
  
extern void cmd_LCD(unsigned char, char); //função externa chamada pelo C  
-----  
void atraso_ms(void) //função chamada pelo assembly  
{  
    _delay_ms(2);  
}  
-----  
void atraso_us(char tempo) //função chamada pelo assembly  
{  
    _delay_us(1);  
    if(!tempo)_delay_us(39);  
}  
-----  
//Sub-rotina de inicialização simplificada para 4 bits de dados do LCD  
-----  
void inic_LCD(void)//envio de instruções para o LCD  
{  
    cmd_LCD(0x28,0); //interface de 4 bits  
    cmd_LCD(0x28,0); //interface de 4 bits 2 linhas (aqui se habilita as 2 linhas)  
    cmd_LCD(0x0c,0); //mensagem aparente cursor inativo não piscando  
    cmd_LCD(0x01,0); //limpa todo o display  
    cmd_LCD(0x80,0); //desloca o cursor para a posição 0x80  
}  
-----  
int main(void)  
{  
    unsigned char i;  
  
    DDRD = 0xFF; //porta D como saída  
    inic_LCD( ); //inicializa o LCD  
  
    for(i=0;(pgm_read_byte(&msg1[i]))!=0;i++)//enviando caractere por caractere  
        cmd_LCD(pgm_read_byte(&msg1[i]),1); //lê na memória flash e usa cmd_LCD  
  
    while(1);//laço infinito  
}
```

cmd_LCD.S (arquivo com a função chamada pelo programa em C)

```
-----  
#define _SFR_ASM_COMPAT 1  
#define __SFR_OFFSET 0  
#include <avr/io.h> //definições do componente especificado  
#define DADOS_LCD PORTD  
#define RS PD2  
#define E PD3
```

```

//-----
//Sub-rotina para enviar comandos ao LCD com dados de 4 bits
//-----
.section .text
.global cmd_LCD ; a função assembly deve ser global
cmd_LCD:
    lds      r16,msg      ;primeiro é interface de 8 bits
for:   push    r24        ;salva dado
       mov     r20,r24    ;copia dado
       in      r24,DADOS_LCD ;lê porta
       andi   r24,0x0F    ;e aplica máscara
       andi   r20, 0xF0    ;aplica máscara no dado
       or     r24,r20    ;compõe para jogar na porta
       out    DADOS_LCD,r24
       sbrs   r22,0       ;verifica se é dado ou instrução
       cbi    DADOS_LCD,RS
       sbrc   r22,0
       sbi    DADOS_LCD,RS
       sbi    DADOS_LCD,E ;pulso de enable > 1 us
       ldi    r24,1
       rcall  atraso_us
       cbi    DADOS_LCD,E
       ldi    r24,0       ;atraso > 40 us entre acessos
       rcall  atraso_us
       pop    r24
       swap   r24        ;prepara para LSB
       dec    r16
       brne   for
       sbrc   r22,0       ;testa se é instrução
       rjmp   sai        ;se é dado sem atraso adicional
       cpi    r24,4       ;verifica qual é a instrução
       brsh   sai        ;para verificar a necessidade de mais atraso
       rcall  atraso_ms
       ldi    r16,2
       sts    msg,r16    ;agora é interface de 4 bits
       ret
.end
//-----

```

20.4 O ARQUIVO MAP

Quando existe a necessidade de otimização de um programa, a solução mais simples é observar a quantidade de bytes ocupados pelo mesmo. Essa informação é facilmente visualizada no AVR Studio. O programador deve estar ciente de como escrever o código (ver otimização no capítulo 4) e, então, basta alterar qualquer linha do programa, recompilá-lo e verificar se sua alteração resultou em menor código.

Para trechos do programa onde o tempo de execução deve ser o menor possível, o programador terá que utilizar a depuração para determinar o

tempo gasto pelo microcontrolador na sua execução. E, caso o tempo seja superior ao desejado, aquele trecho em específico precisa ser alterado.

Quando existir a necessidade de maiores detalhes quanto a número de bytes consumidos pelo programa, tal como o que cada função ou parte do programa ocupa, o programador deve consultar um arquivo criado pelo compilador, com extensão **map**. Além da informação do número de bytes ocupados pelas diferentes partes do programa, o arquivo *.map também fornece outras informações como, por exemplo, o endereço de gravação do código na memória de programa.

Dentro da pasta de projeto criada no AVR Studio, existirá uma pasta chamada **Debug**, onde o arquivo *.map pode ser encontrado. A desvantagem do uso desse arquivo é o formato das informações gravadas, o que torna difícil a interpretação das mesmas.

Exercícios:

20.1 – Criar um programa em *assembly* que chame uma função em C.

20.2 – Criar um programa em C que chame uma função em *assembly*. Como colocar um trecho de código *assembly* dentro de um código em C?

20.3 – Quais são as técnicas utilizadas para a otimização de um programa em C?

20.4 – Abra um arquivo *.map de um projeto que você já fez e o analise.

21. RTOS

Este capítulo trata de uma importante ferramenta de programação para desenvolvimento de projetos de mais alto nível, os sistemas operacionais de tempo real - RTOS, compatíveis principalmente com arquiteturas de 32 bits. Entretanto, com o desenvolvimento de novos *firmwares* e com a maior disponibilidade de memória, os microcontroladores de 8 bits podem suportar sistemas operacionais mínimos e ainda apresentar desempenho satisfatório. A utilização de um RTOS se justifica pelas facilidade de programação, permitindo a solução de problemas complexos, difíceis de tratar com a programação convencional. Assim, com algum sacrifício de desempenho, pode-se utilizar um RTOS em um microcontrolador de 8 bits.

21.1 INTRODUÇÃO

Um Sistema Operacional (SO) pode ser definido como uma coleção de programas que atua como uma interface entre os programas do usuário e o hardware. Sua principal função é proporcionar um ambiente para que o usuário de um sistema microprocessado execute programas no hardware do sistema de uma maneira conveniente e eficiente. Em resumo, as tarefas próprias de um sistema operacional são:

- Gerenciar o processador, a memória e os dispositivos de I/O.
- Oferecer uma interface simples para as aplicações e para o usuário.

Com base no número de usuários permitidos e no modo de execução, um sistema operacional pode ser classificado como:

- Monousuário, monotarefa: permite que um usuário possa realizar uma atividade de cada vez. O antigo MS-DOS é um exemplo de um

sistema operacional monousuário, monotarefa.

- Monousuário, multitarefa: o mais utilizado em computadores de uso pessoal. O Windows 7 e o MacOS são exemplos de um SO que permite que um único usuário interaja simultaneamente com vários programas.
- Multiusuário: permite que diversos usuários possam tirar simultaneamente vantagem dos recursos do computador. O sistema operacional deve equilibrar as exigências de todos os usuários e se certificar de que cada um dos programas utilizados contam com recursos separados e suficientes, para que um problema com um usuário não afete os outros usuários. O Unix é um exemplo de sistema operacional multiusuário.
- Sistema Operacional de Tempo Real (RTOS): utilizado em computadores embarcados em robôs, instrumentos científicos e sistemas industriais. Tipicamente, um RTOS tem pouca capacidade de interação com o usuário e nenhuma utilidade para o usuário final, já que o sistema se comportará como uma ‘caixa preta’ quando disponibilizado para uso. Sua principal tarefa é gerenciar os recursos do computador, de tal forma que uma determinada operação seja executada na mesma fração de tempo que as demais em andamento.

A principal diferença entre os sistemas operacionais multitarefas de propósito geral (GPOS) e os sistemas operacionais de tempo real é a necessidade dos RTOS apresentarem um comportamento temporal previsível, ou seja, os serviços do sistema operacional devem ser executados em frações conhecidas e esperadas de tempo.

Os serviços dos GPOS podem produzir atrasos aleatórios no software aplicativo, o que pode causar, em momentos inesperados, a resposta lenta de um aplicativo. O Comportamento temporal previsível não é um objetivo de projeto dos GPOS, eles são desenvolvidos para garantir que todos os

processos tenham a possibilidade de serem executados pela CPU. Não há preocupação com a previsibilidade temporal. O objetivo de um GPOS é garantir um tempo médio de execução.

Os RTOS, por sua vez, oferecem um tempo de execução constante independente da carga para a maioria dos serviços da CPU. Por exemplo, o tempo de transmissão de uma mensagem por uma tarefa é constante, independente de fatores, tais como: o tamanho da mensagem a ser enviada, o número de tarefas e filas de mensagens que estão sendo gerenciados pelo RTOS.

Os RTOS, em resumo, são concebidos para operar sistemas que devem prestar os seus serviços em prazos de tempo restritos e esperados e que possuem capacidade de memória e poder de processamento limitados. O núcleo (*kernel*) de um RTOS fornece uma camada de abstração cujo objetivo é esconder do usuário os detalhes do hardware do processador sob o qual o seu software será executado, conforme ilustrado na fig. 21.1. O software do usuário é dividido em blocos chamados tarefas (*tasks*). As tarefas, por sua vez, possuem restrições temporais e uma relação de comunicação e sincronização com outras tarefas.

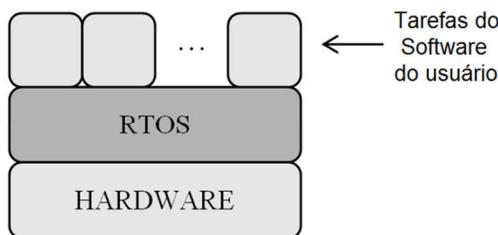


Fig. 21.1 - O núcleo do RTOS (*kernel*) fornece uma camada de abstração entre o software do usuário e o hardware.

A partir da camada de abstração fornecida, o núcleo do RTOS oferece cinco principais categorias de serviços básicos para o software do usuário, conforme ilustrado na fig. 21.2.



Fig. 21.2 – Serviços básicos oferecidos pelo núcleo de um RTOS.

A Gestão de Tarefas, representada no centro da fig. 21.2, é a principal categoria de serviço do núcleo. O conjunto de serviços oferecidos pelo RTOS permite que um aplicativo possa ser implementado, testado e mantido mais facilmente através da divisão do problema a ser resolvido em tarefas menores e de compreensão mais fácil. Essa abordagem modular permite, ainda, que as tarefas individuais sejam utilizadas em outros projetos. Nessa categoria, o principal serviço do RTOS é o escalonador de tarefas, serviço que controla a execução das tarefas da aplicação.

Outra categoria de serviços do núcleo apresentada na fig. 21.2 é a sincronização e comunicação entre tarefas. Esses serviços permitem que tarefas troquem informações sem o risco de que sejam corrompidas.

Em função dos prazos rigorosos de tempo que devem ser cumpridos pelas tarefas em muitos sistemas embarcados, os RTOS geralmente disponibilizam serviços básicos de temporização, tais como serviços de atraso e de limite de tempo.

Outros serviços que podem ser oferecidos por um RTOS são a alocação dinâmica de memória e a supervisão de dispositivos de I/O. Os serviços de supervisão de dispositivos de I/O fornecem suporte para organizar e

acessar os *drivers* do hardware de um sistema embarcado. A alocação dinâmica de memória permite que as tarefas solicitem porções da memória RAM para uso temporário no software aplicativo. Esse serviço não é oferecida pelos núcleos de pequenos RTOS concebidos para microcontroladores com limitação de memória.

Ainda, um RTOS pode expandir seus serviços oferecendo componentes opcionais, tais como: a organização de sistema de arquivos, gerenciamento de rede e interface gráfica de usuário. Buscando minimizar o consumo de memória de programa, os componentes de expansão são incluídos no sistema embarcado somente se os seus serviços forem necessários para a implementação da aplicação.

21.2 GESTÃO DE TAREFAS

Em um sistemas monotarefas, representado pelo diagrama de estados da fig. 21.3, o processador fica ocioso enquanto aguarda por um recurso de hardware mais lento como, por exemplo, uma leitura em disco. Em um sistema multitarefas, esse problema é resolvido permitindo que o processador suspenda a execução da tarefa que aguarda por dados externos e passe a executar outra tarefa. Um sistema operacional capaz de retirar uma tarefa da CPU antes do término da sua execução é chamado de preemptivo. A preempção torna os sistemas mais produtivos, permitindo que várias tarefas possam estar em andamento ao mesmo tempo, porém em estados diversos. Um sistema com essa capacidade é chamado multitarefa, podendo ser representado pelo diagrama de estados da fig. 21.4.



Fig. 21.3 – Diagrama de estados de um sistema monotarefa.

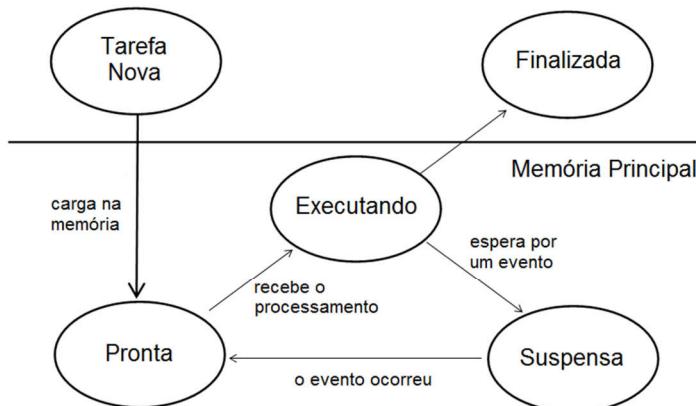


Fig. 21.4 – Diagrama de estados das tarefas em um sistema multitarefa.

Em um sistema multitarefa, as tarefas podem assumir cinco estados:

- Tarefa nova - o sistema operacional está ciente da sua existência, porém não lhe foi alocada uma prioridade e um contexto.
- Pronta - a tarefa está aguardando para ser processada em uma fila cuja ordem de execução é definida por um algoritmo de escalonamento.
- Executando – neste momento, o processador está dedicado à tarefa, executando seu código. Ela ficará em execução até que precise de algum dado externo ou aguardar por um evento ou, ainda, até que uma tarefa de maior prioridade receba o processamento no seu lugar.
- Suspensa - a tarefa não pode ser executada porque precisa de algum dado externo não disponibilizado, esperar por um evento ou que outra tarefa libere o processador.
- Finalizada - o processamento da tarefa foi encerrado e a memória destinada a sua estrutura é liberada.

Quando uma tarefa é gerada, ou pelo sistema operacional, ou por outra tarefa, inicia-se um processo que envolve carregá-la na memória, criar e atualizar certas estruturas de dados do sistema operacional necessárias para a sua execução. Até terminar esse processo, a tarefa está no estado de ‘nova’. Uma vez que o processo se encerra, ela entra no estado ‘pronta’, onde fica aguardando para ser processada. Neste momento, a sua execução passa a ser considerada pelo escalonador (*task scheduler*), que é quem decide a ordem de execução das tarefas prontas.

Uma tarefa passa do estado ‘pronta’ para o estado ‘executando’ pela ação do despachante ou executivo (*dispatcher*) quando o escalonador determina qual tarefa deve ser executada, de acordo com sua política de agendamento. Quando a tarefa encerra o seu processamento, ela entra no estado ‘finalizada’ e, em seguida, é removida da visão do sistema operacional.

Durante a execução da tarefa, podem ocorrer interrupções de software ou hardware. Nesse caso, dependendo da prioridade da interrupção, a tarefa atual pode ser transferida para o estado pronta e esperar pela sua próxima alocação pelo escalonador. Finalmente, uma tarefa pode, durante o seu curso de execução, ser interrompida devido a exigências de sincronização com outras tarefas, ou para aguardar a conclusão de alguns serviços que ela requisitou. Durante tal tempo, ela está no estado ‘suspenso’. Uma vez que o requisito de sincronização é cumprido, ou o serviço solicitado é concluído, ela é retornada para o estado ‘pronta’, passando a esperar o seu próximo agendamento.

Sempre que ocorre uma troca de tarefas, o contexto da tarefa em execução, representado pelos conteúdos do contador de programa, pilha e registradores, é salvo pelo sistema operacional em uma estrutura especial de dados, chamada de bloco de controle da tarefa (*Task Control Block – TCB*), de modo que a tarefa possa recomeçar em seu próximo agendamento. Por sua vez, o contexto da tarefa que entrará em execução tem que ser restaurado a partir do seu TCB, antes de receber o

processador. Dessa forma, sempre é perdido tempo de processamento durante a troca de contexto, no qual nenhuma tarefa está sendo executada. O diagrama temporal que representa uma troca de contexto entre duas tarefas é mostrado na fig. 21.5.

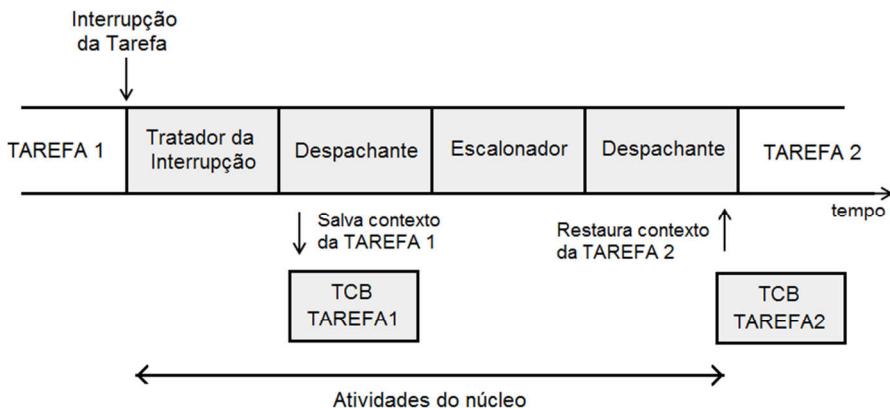


Fig. 21.5 – Diagrama temporal da troca de contexto entre duas tarefas. As atividades em cinza são realizadas pelo núcleo do sistema operacional.

Os escalonadores da maioria dos RTOSs usam um esquema conhecido como escalonamento preemptivo baseado em prioridade. No escalonamento por prioridades, uma prioridade é atribuída para cada tarefa da aplicação, geralmente na forma de um número inteiro, sendo que quanto mais rápida for a resposta desejada para a tarefa, maior será a prioridade que lhe deve ser atribuída. É a natureza preemptiva do escalonador de tarefas que garante a rapidez desejada na resposta. O termo preemptivo significa que o escalonador pode, a qualquer momento, parar a execução de uma tarefa se ele determinar que outra tarefa precisa imediatamente receber o processador. Assim, para o escalonador preemptivo baseado em prioridade, em cada instante de tempo, a tarefa pronta para executar e de maior prioridade é a tarefa que deve estar sendo executada. Ou seja, se uma tarefa de baixa prioridade e uma tarefa da alta prioridade estão prontas para serem executadas, o escalonador permitirá

que a tarefa de alta prioridade seja executada primeiro. A tarefa de baixa prioridade só conseguirá ser executada após a tarefa de alta prioridade ser finalizada.

Considerando-se três tarefas: uma de baixa prioridade, uma de média prioridade e outra de alta prioridade. Se uma tarefa de maior prioridade (alta ou média) tornar-se pronta quando uma tarefa de prioridade baixa estiver em execução, o escalonador preemptivo retirará a tarefa de baixa prioridade de funcionamento após ela completar a instrução em linguagem *assembly* em execução e entregará o processador para a tarefa de maior prioridade. Após a tarefa de mais alta prioridade finalizar o seu trabalho, a tarefa de baixa prioridade ganhará novamente o controle do processador. Obviamente, enquanto a tarefa de média prioridade está em execução, o processo de alta prioridade pode ficar pronto. Nesse caso, a tarefa de média prioridade será interrompida para permitir que a tarefa de alta prioridade seja executada. Quando a tarefa de alta prioridade terminar o seu trabalho, a tarefa de média prioridade reassume o processador. Essa situação poderia ser chamada de aninhamento preemptivo. O exemplo de uma linha do tempo representando o escalonamento preemptivo baseado em prioridades é mostrada na fig. 21.6.

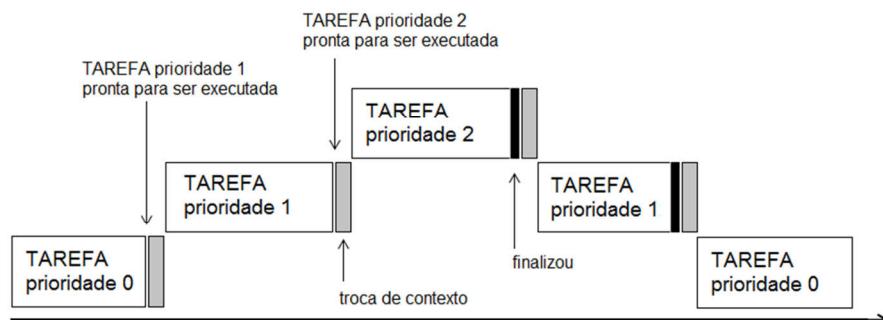


Fig. 21.6 – Exemplo de uma linha do tempo para o escalonamento preemptivo baseado em prioridades (o número 2 representa a maior prioridade).

Um RTOS menos complexo pode fazer a troca de tarefas fornecendo um tempo limite de processamento para cada tarefa, o quantum. Esgotado esse tempo, a tarefa em execução perde o processador e volta para a fila de tarefas prontas. Essa ‘preempção por limite de tempo’ é realizada pela interrupção por estouro de um temporizador do hardware. Em resumo, cada tarefa é executada por um período fixo de tempo até ser finalizada. Então, se a necessidade de uma mudança de tarefa surgisse em qualquer lugar dentro da janela do quantum, a troca de tarefa ocorreria apenas no final do tempo limite. Esse atraso seria inaceitável na maioria dos sistemas embarcados de tempo real. Um exemplo de tarefas com tempo limite para execução é apresentado na fig. 21.7.

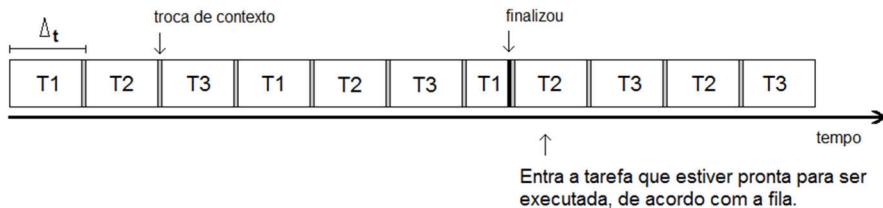


Fig. 21.7 – Diagrama de estados das tarefas em um sistema multitarefa, com tempo limite para execução das tarefas (tarefas T1, T2 e T3 todas com a mesma prioridade).

Como pode ser percebido na fig. 21.7, cada tarefa possui o mesmo tempo de execução. Dessa forma, pode-se calcular quantas trocas de tarefas são feitas por segundo. Como as tarefas são todas executadas rapidamente uma após a outra, o sistema operacional dá a impressão que todas estão sendo executadas em paralelo. Dependendo do sistema operacional, caso uma tarefa tenha sido concluída e ainda não exista uma tarefa pronta para ser executada, é possível que ele execute uma tarefa específica para essa ocasião (*Idle*).

21.3 COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE TAREFAS

Os sistemas operacionais oferecem mecanismos para a comunicação e a sincronização entre tarefas. Tais mecanismos são necessários em um ambiente preemptivo multitarefas, porque na ausência deles, as tarefas podem transmitir informações corrompidas ou interferirem umas nas outras. Por exemplo, uma tarefa pode perder o processador quando está no meio da atualização de uma tabela de dados. Se uma segunda tarefa, que adquiriu o processador, ler a partir dessa tabela, ela lerá uma combinação de algumas áreas de dados recém atualizadas e de outras que ainda não foram atualizadas. Essa atualização parcial dos dados deve torná-los inconsistentes.

Provavelmente, a forma mais popular de comunicação entre tarefas em sistemas embarcados é a transmissão de dados de uma tarefa para outra. Muitos RTOS oferecem um mecanismo de transmissão de mensagens para esse fim, conforme ilustrado na fig. 21.8. Cada mensagem pode conter uma matriz ou um registrador de dados (*buffer*). Se as mensagens podem ser enviadas mais rapidamente do que podem ser tratadas, o RTOS fornecerá filas de mensagens para manter as mensagens até que elas possam ser processadas.

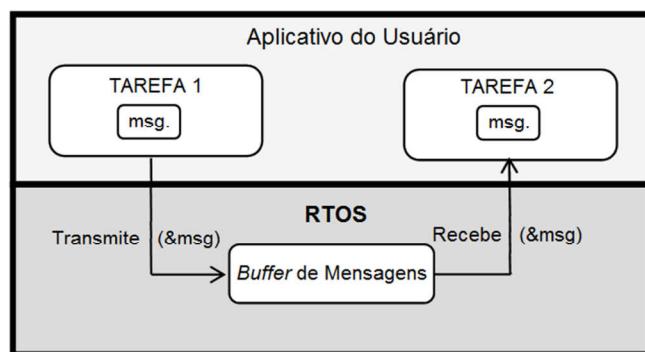


Fig. 21.8 – Esquema da comunicação entre tarefas através de mensagens.

Através do conceito de *mailbox* (caixa de correio) mensagens podem ser trocadas sem conflitos entre tarefas. As caixas podem ser usadas livremente por todas as tarefas e interrupções. Elas são identificadas com um número de *mailbox*. Geralmente, as caixas de correio permitem as seguintes operações:

- Enviar uma mensagem - cada tarefa pode enviar uma mensagem para qualquer caixa de correio. Neste caso, a mensagem a ser enviada é copiada na lista de mensagens. Se a lista de mensagens da caixa de correio já está cheia durante o envio, a tarefa é colocada no estado de espera (entrou na lista de espera de escrita). Ela permanece no estado de espera até outra tarefa buscar uma mensagem na caixa e, então, gerar espaço. Como alternativa, um tempo limite pode ser especificado para o envio, após o qual a espera é abortada. Se a lista de mensagens não está cheia quando o envio ocorre, a mensagem é imediatamente copiada na lista de mensagem e a tarefa não deve esperar.
- Ler uma mensagem - cada tarefa pode ler uma mensagem de uma caixa de correio qualquer. Se a lista de mensagens da caixa está no momento vazia (nenhuma mensagem disponível), a tarefa é colocada no estado de espera (entrou na lista de espera de leitura). Ela permanece no estado de leitura até outra tarefa enviar uma mensagem para a caixa. Como uma alternativa, um tempo limite pode ser especificado para a leitura após o qual a espera será abortada. Se a lista de mensagens não está vazia no momento da leitura, então a tarefa recebe imediatamente a mensagem.

Outra espécie de comunicação entre tarefas em sistemas embarcados é a transmissão do que pode ser chamada de informação de sincronização de uma tarefa para outra. Essa informação de sincronização pode ser considerada um comando, que por sua vez pode ser negativo ou positivo. Um comando negativo ocorre quando há concorrência entre as tarefas, ou seja, as tarefas requerem o uso de algum recurso que não pode ser

utilizado simultaneamente por mais de uma tarefa. Um exemplo seria o bloqueio, por parte de uma tarefa, do uso da USART do sistema para o seu próprio uso, impedindo a sua utilização por outra tarefa que solicitar o acesso a esse periférico. Um comando positivo ocorre quando há cooperação entre tarefas, ou seja, uma determinada tarefa precisa aguardar que outra tarefa conclua alguma operação específica para que ela possa prosseguir a sua execução.

Muitos RTOS oferecem um mecanismo de semáforo para o tratamento da sincronização negativa. Esse mecanismo permite que as tarefas bloqueiem certos recursos do sistema embarcado apenas para o seu uso, recurso que seria liberado após a sua utilização. Um semáforo contém um passe (*token*) que o código adquire para continuar a execução. Se o recurso já está em uso, a tarefa requerente é bloqueada até o passe ser retornado ao semáforo pela sua concorrente. Existem dois tipos de semáforos:

- Binários – permitem somente dois valores: zero (indisponível), um (disponível); resolvem o problema da exclusão mútua.
- Contadores – permitem uma gama maior de valores. Neste caso, o semáforo é um contêiner que mantém um número de passes. Antes de uma tarefa continuar, ela deve adquirir um passe para executar seu procedimento e, então, retornar o passe. Se todos os passes foram adquiridos por outras tarefas, a tarefa requerente esperará até outra tarefa retornar um passe para o semáforo.

Os semáforos geralmente permitem as seguintes operações:

- Espera por passe – quando uma tarefa requisita por meio de uma função do RTOS um recurso controlado por um semáforo, se um passe está disponível a tarefa continuará a sua execução. De outra forma, ela será bloqueada até o passe ser disponibilizado ou, em alguns casos, um tempo limite opcional for excedido.
- Retorna (envia) passe - após completar a sua operação sobre um recurso, a tarefa retornará o passe associado ao semáforo através de uma função do RTOS.

O termo semáforo binário é usado frequentemente como um sinônimo de mutex. Porém, em alguns RTOS, ao contrário dos semáforos, apenas uma tarefa é proprietária de um mutex num determinado momento. Ou seja, um mutex pode ser destravado somente pela tarefa que se apropriou dele.

Para a sincronização positiva, diferentes RTOS oferecem diferentes mecanismos. Para a sincronização entre as tarefas sem a troca de dados, alguns RTOS oferecem os sinais. Nesse caso, cada tarefa ativa conteria seu próprio bit de sinal com o qual as seguintes operações poderiam ser executadas:

- Espera por sinal - cada tarefa pode esperar por seu bit de sinal. Ela espera até seu bit de sinal ser ativado por outra tarefa. Após o sinal ser recebido, a tarefa em espera limpa seu bit de sinal e entra para o estado ‘pronta’ ou ‘executando’.
- Envia um sinal - cada tarefa e cada interrupção podem ativar um bit de sinal de qualquer outra tarefa.
- Limpa um sinal - uma tarefa pode limpar o bit de sinal de qualquer outra tarefa.

21.4 ALOCAÇÃO DINÂMICA DE MEMÓRIA

Muitos sistemas operacionais de computação geral em tempo não real oferecem serviços de alocação de memória a partir da memória livre (*heap*) que se encontra entre a área destinada ao armazenamento permanente e a pilha. Os serviços *malloc* e *free*, utilizados na linguagem C, trabalham a partir dessa memória livre. Chamando *malloc*, uma tarefa solicita ao sistema operacional um pedaço desta memória livre. Quando essa tarefa, ou mesmo outra tarefa, não necessitar mais da memória alocada, ela pode liberá-la chamando *free*, permitindo que o sistema operacional possa destiná-la para a utilização em outra tarefa.

A memória livre está sujeita a um processo de fragmentação que pode causar a degradação do serviço de alocação. Essa fragmentação é causada porque quando a área de memória solicitada é liberada, ela pode ser dividida em porções menores pelas próximas chamadas de *malloc*. Após muitas chamadas de *malloc* e *free*, pequenas porções de memória podem aparecer entre as áreas da memória livre que estão sendo usados pelas tarefas. Essas porções, inúteis para as tarefas, ficam presas entre as áreas de memória em uso, não podendo ser aglutinadas em uma porção maior. Ao longo do tempo, a memória livre estará repleta destas porções, o que pode gerar a recusa por parte do sistema operacional da solicitação de memória de certo tamanho, apesar do sistema operacional ter memória suficiente disponível. Em outras palavras, a memória disponível está dispersa em pequenos fragmentos distribuídos em várias partes separadas da memória livre e o sistema operacional não pode atender a solicitação de uma tarefa. A fragmentação pode ser resolvida por softwares de desfragmentação. Contudo, esses softwares injetam atrasos de duração aleatória nos serviços da memória livre.

Os sistemas operacionais de tempo real não podem conviver com atrasos aleatórios, nem permitir que a fragmentação de sua memória faça com que eles recusem o uso da memória livre por uma tarefa. Eles evitam a fragmentação da memória utilizando blocos de dimensões fixas, os *pools*. Um RTOS permite configurar vários *pools*, cada um consistindo do mesmo número de porções de memória, os *buffers*. Em um dado *pool*, todos os *buffers* são do mesmo tamanho. Os *pools* evitam a fragmentação da memória externa, pois não permitem que um *buffer* retornado ao *pool* seja dividido em *buffers* menores em futuras solicitações. Em vez disso, quando um *buffer* é retornado ao *pool*, ele é colocado em uma ‘lista de *buffers* livres’ de *buffers* do seu tamanho, que ficam disponíveis para futura reutilização no seu tamanho original. Através desse mecanismo de alocação, a memória é alocada e liberada a partir do *pool* em um tempo previsível e constante.

21.5 BRTOS

Para ilustrar o funcionamento de um RTOS será utilizado o *Brazilian Real-Time Operating System* (BRTOS), desenvolvido por brasileiros e gratuito. O BRTOS é um sistema operacional de tempo real de pequeno porte que suporta 32 tarefas e fornece controle sobre semáforos, mutex, caixas de mensagens e filas. Ele executa o escalonamento preemptivo baseado em prioridades e conta com suporte para diversos microcontroladores, entre eles o ATmega328.

Para utilizar o BRTOS em uma aplicação, é necessário configurar alguns parâmetros do sistema no seu arquivo **BRTOSConfig.h**:

- Para evitar o uso desnecessário da memória do sistema, é importante definir o número de tarefas (NUMBER_OF_TASKS) que serão utilizadas na aplicação.
- Para definir a resolução do gerenciador de tempo do RTOS (marca de tempo), é preciso definir a frequência do barramento do microcontrolador em Hz (configCPU_CLOCK_HZ), a resolução do gerenciador de tempo do RTOS propriamente dita (configTICK_RATE_HZ), cujos valores recomendados estão entre 1 ms (1000 Hz) e 10 ms (100 Hz), e o *prescaler* do periférico responsável pela base de tempo (configTIMER_PRE_SCALER e configTIMER_PRE_SCALER_VALUE). Esses valores são utilizados pela função void TickTimerSetup(void) do arquivo **hal.c**.
- Para cada serviço utilizado é preciso habilitá-lo e definir o seu número máximo de instâncias. Por exemplo, se a aplicação utilizar o serviço de semáforos, é preciso definir o valor 1 para BRTOS_SEM_EN e um valor para BRTOS_MAX_SEM.

A seguir, é apresentada a estrutura do arquivo **BRTOSconfig.h** para o microcontrolador Atmega328 utilizada nos dois próximos exemplos, com a frequência da CPU ajustada em 16 MHz e o passo de tempo em 1 ms.

```

// Define if simulation or DEBUG
#define DEBUG 1

/// Define if verbose info is available
#define VERBOSE 0

/// Define if error check is available
#define ERROR_CHECK 0

/// Define if compute cpu load is active
#define COMPUTES_CPU_LOAD 1

// Define if whatchdog active
#define WATCHDOG 1

/// Define Number of Priorities
#define NUMBER_OF_PRIORITIES 16

/// Define if OS Trace is active
#define OSTRACE 0

#if (OSTRACE == 1)
    #include "debug_stack.h"
#endif

// Define the number of Task to be Installed
// must always be equal or higher to NumberOfInstalledTasks
#define NUMBER_OF_TASKS 6

/// Define if TimerHook function is active
#define TIMER_HOOK_EN 0

/// Define if IdleHook function is active
#define IDLE_HOOK_EN 0

// Habilita o serviço de semáforo do sistema
#define BRTOS_SEM_EN           1

// Habilita o serviço de mutex do sistema
#define BRTOS_MUTEX_EN          1

// Habilita o serviço de mailbox do sistema
#define BRTOS_MBOX_EN            0

// Habilita o serviço de filas do sistema
#define BRTOS_QUEUE_EN           1

/// Enable or disable queue 16 bits controls
#define BRTOS_QUEUE_16_EN         0

/// Enable or disable queue 32 bits controls
#define BRTOS_QUEUE_32_EN         0

// Define o número máximo de semáforos (limita a alocação de memória p/ semáforos)
#define BRTOS_MAX_SEM             2

// Define o número máximo de mutex (limita a alocação de memória p/ mutex)
#define BRTOS_MAX_MUTEX            2

// Define o número máximo de Mailbox (limita a alocação de memória p/ mailbox)
#define BRTOS_MAX_MBOX              1

```

```

// Define o número máximo de filas (limita a alocação de memória p/ filas)
#define BRTOS_MAX_QUEUE           3

// TickTimer Defines
#define configCPU_CLOCK_HZ        (INT32U)16000000
#define configTICK_RATE_HZ         (INT32U)1000
#define configTIMER_PRE_SCALER    (INT8U)3
#define configTIMER_PRE_SCALER_VALUE (INT8U)64
#define OSRTCEN                   0

//Stack Defines
// P/ ATMEGA com 2KB de RAM, configurado com 512 p/ STACK Virtual
#define HEAP_SIZE 4*128

// Queue heap defines
// Configurado com 512B p/ filas
#define QUEUE_HEAP_SIZE 1*32

// Stack Size of the Idle Task
#define IDLE_STACK_SIZE           (INT16U)80

```

O BRTOS utiliza o temporizador/contador 0 do ATmega328 para gerar a base de tempo (*time tick*) do sistema operacional. Desta forma, o software de aplicação não pode empregar esse temporizador.

Para ilustrar a utilização do BRTOS, será empregado um exemplo bem simples, um contador de dois dígitos hexadecimal, com as seguintes características:

- um botão para contagem crescente (0 a 0xFF);
- um botão para contagem decrescente (0xFF a 0);
- um botão para a inicialização da contagem (0x00);
- dois *displays* de 7 segmentos multiplexados.

Cada vez que um botão de contagem é pressionado ou mantido pressionado a contagem é efetuada. Quando a contagem chega ao seu valor máximo, volta novamente ao mínimo, e vice-versa.

O BRTOS permite uma abordagem modular do problema, onde cada função do programa será desempenhada por uma tarefa. Na tab. 21.1, é apresentado um resumo comparativo entre a programação convencional e a programação com o auxílio de um RTOS para a implementação do exemplo supracitado.

Tab. 21.1 – Exemplo comparativo entre a programação convencional e a programação empregando um RTOS.

Programação Convencional	RTOS
<pre> int main() { Inicializações_normais(); while(1) { Multiplexa_Display(); Incrementa_Contagem(); Decrementa_Contagem(); Inicializa_Contagem(); } } //----- //funções do programa //----- void Multiplexa_Display() {.....}; void Incrementa_Contagem(){.....}; void Decrementa_Contagem(){.....}; void Inicializa_Contagem(){.....}; </pre>	<pre> int main() { Inicializações_normais(); Inicializações_RTOS(); while(1); } //----- //tarefas do RTOS //----- void Multiplexa_Display() {while(1){...}}; void Incrementa_Contagem(){while(1){...}}; void Decrementa_Contagem(){while(1){...}}; void Inicializa_Contagem(){while(1){...}}; </pre>
Pode utilizar funções ou não. Cada função não pode conter um laço infinito.	Cada função é uma tarefa independente do RTOS, com um laço infinito de execução.
As tarefas são sequenciais, executadas no programa principal dentro de um laço infinito. O paralelismo que se consegue é com o uso das interrupções do microcontrolador.	As tarefas são concorrente. O RTOS gerencia sua execução. Existe a sensação de paralelismo na execução e, ainda, as interrupções normais do microcontroladores podem ser utilizadas.
O programa pode travar em alguma função, como por exemplo, quando existe um laço de espera que nunca é finalizado, pois a execução do programa é sequencial.	O programa não trava em funções com laços de espera. Se uma tarefa ficar presa, o RTOS vai passar o controle para outra função com o passar do tempo.
Dependendo do problema, maior complexidade de programação.	Menor complexidade de programação – melhor estruturação.
Uso de menor quantidade de memória.	Emprega mais memória, impacto maior na RAM.
Pode obter o máximo de desempenho, com a utilização constante da CPU para a execução do programa.	Perda de processamento na troca de contexto pela CPU (overhead).

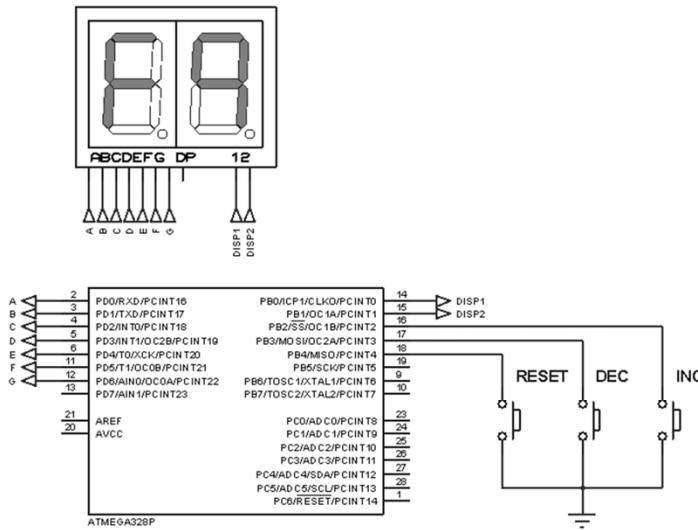


Fig. 21.9 – Diagrama esquemático do contador que embarca o BRTOS no seu firmware (circuito simplificado).

Para a execução do exemplo acima, são necessários os seguintes passos:

1. Baixar a versão mais recente do BRTOS para o ATmega328 (<http://code.google.com/p/brtos/downloads/list>) e descompactar em uma pasta de livre escolha. Neste exemplo, foi utilizado o arquivo **BRTOS 1.6x - ATMEGA328.rar**.
2. Abrir o AVR Studio e criar o projeto com o nome Exemplo_BRTOS na pasta desejada, com a seleção do ATmega328P.
3. Copiar na pasta do projeto <..\Exemplo_BRTOS\Exemplo_BRTOS>, onde fica o arquivo **Exemplo_BRTOS.c** (principal), somente as pastas geradas na descompactação do arquivo do BRTOS: brtos, drivers, hal e includes; não incluir os arquivos **main.c** e demais, que são do projeto exemplo do BRTOS.
4. No arquivo **Exemplo_BRTOS.c** coloque o código apresentado abaixo (programa principal).

```
#include "BRTOS.h"
#include "tasks.h"
#define FOSC 16000000 // Frequência da CPU

// Associa nomes das tarefas
const CHAR8 Task1Name[] PROGMEM = "Multiplexa_Display";
const CHAR8 Task2Name[] PROGMEM = "Incrementa_Contagem";
const CHAR8 Task3Name[] PROGMEM = "Decrementa_Contagem";
```

```

const CHAR8 Task4Name[] PROGMEM = "Inicializa_Contagem";

PGM_P MainStringTable[] PROGMEM =
{
    Task1Name,
    Task2Name,
    Task3Name,
    Task4Name
};

//-----
void avr_Init() //inicialização dos registradores do microcontrolador utilizado
{
    DDRD = 0xFF;
    PORTD = 0xFF;
    DDRB = 0b00000011;
    PORTB = 0b11111100;
}
//-----
int main(void)
{
    avr_Init();      //inicialização dos registradores do microcontrolador utilizado
    BRTOS_Init();   //inicialização do BRTOS

    /* Instala todas as tarefas no seguinte formato: (Endereço da tarefa, Nome da
       tarefa, Número de Bytes do Stack Virtual, Prioridade da Tarefa)
       Cada tarefa deve possuir uma prioridade. A instalação de uma
       tarefa em um prioridade ocupada gerará um código de exceção*/
    if(InstallTask(&Multiplexa_Display,Task1Name,40,4) != OK)//tarefa de maior prioridade
        while(1){};// Oh Oh - Não deveria entrar aqui !!!

    if(InstallTask(&Incrementa_Contagem,Task2Name,40,2) != OK)
        while(1){};// Oh Oh - Não deveria entrar aqui !!!

    if(InstallTask(&Decrementa_Contagem,Task3Name,40,1) != OK)
        while(1){};// Oh Oh - Não deveria entrar aqui !!!

    if(InstallTask(&Inicializa_Contagem,Task4Name,40,3) != OK)
        while(1){};// Oh Oh - Não deveria entrar aqui !!!

    /*Inicialização do escalonador. A partir deste momento as tarefas instaladas
       começam a ser executadas*/
    if(BRTOSStart() != OK)
        while(1){};// Oh Oh - Não deveria entrar aqui !!!

    while(1{});//laço infinito
}
//-----

```

5. Adicionar ao projeto os arquivo **BRTOS.c** e **HAL.c**, respectivamente, das pastas: <..\\Exemplo_BRTOS\\Exemplo_BRTOS\\brtos> e <..\\Exemplo_BRTOS\\Exemplo_BRTOS\\hal> .
6. Adicionar ao projeto um novo arquivo, renomeá-lo como **tasks.c** e inserir o código a seguir (é neste arquivo que estão as tarefas).

```

#include "BRTOS.h"
#include "drivers.h"
#include "tasks.h"

#define DEBOUNCE 10
#define MUX 2

#define botao_INCREMENTO !(PINB&(1<<PINB2)) //leitura do botão de incremento
#define botao_DECREMENTO !(PINB&(1<<PINB3)) //leitura do botão de decremento
#define botao_RESET !(PINB&(1<<PINB4)) //leitura do botão de reset

const char tabela[16] PROGMEM = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12, 0x02, 0x78, 0x00,
                                0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E};

volatile unsigned char cont;
//-----
void Multiplexa_Display()
{
    while(1)
    {
        PORTB &= 0xFD; //desliga display 2
        PORTD = pgm_read_byte(&tabela[cont/16]); //valor no display 1
        PORTB |= 0x01; //liga display 1
        DelayTask(MUX); //espera por intervalo: MUX ticks
        PORTD &= 0xFE; //desliga display 1
        PORTD = pgm_read_byte(&tabela[cont%16]); //valor no display 2
        PORTB |= 0x02; //liga display 2
        DelayTask(MUX); //espera por intervalo: MUX ticks
    }
}
//-----
void Incrementa_Contagem()
{
    while(1)
    {
        if(botao_INCREMENTO)
        {
            cont++;
            DelayTask(200);
        }
        else
            DelayTask(1);
    }
}
//-----
void Decrementa_Contagem()
{
    while(1)
    {
        if(botao_DECREMENTO)
        {
            cont--;
            DelayTask(200); //aguarda expirar o tempo: DEBOUNCE ticks
        }
        else
            DelayTask(1);
    }
}
//-----

```

```

void Inicializa_Contagem(void)
{
    while(1)
    {
        if(botao_RESET)
            cont=0;
        else
            DelayTask(1);
    }
}
//-----

```

7. Modificar o arquivo **tasks.h** na pasta

<...\\Exemplo_BRTOS\\Exemplo_BRTOS\\includes>, substituindo o código existente pelo abaixo.

```

/****************************************************************************
 *                                BRTOS
 *          Brazilian Real-Time Operating System
 *          Acronymous of Basic Real-Time Operating System
 *          Open Source RTOS under MIT License
 * OS Tasks
 ****
#include "hardware.h"
void Multiplexa_Display(void);
void Incrementa_Contagem(void);
void Decrementa_Contagem(void);
void Inicializa_Contagem(void);
void Transmite_Uptime(void);
void Transmite_Duty_Cycle(void);
void Transmite_RAM_Ocupada(void);
void Transmite_Task_Stacks(void);
void Transmite_CPU_Load(void);
void Reason_of_Reset(void);

```

8. Acessar o menu <Project> <Properties> <toolchain> <AVR/GNU C Compiler> <Directories>, caso essa opção não esteja disponível é possível clicar diretamente no ícone da janela principal  ATmega328P, ver a fig. 3.12 (capítulo 3). Então, devem ser adicionados os caminhos dos *includes* do BRTOS (*Include Paths*), os quais são:

-/brtos/includes
-/hal
-/drivers
-/includes

9. Escolher o nível de otimização desejado (fig. 3.12, capítulo 3).

10. Finalmente, compilar o projeto.

A seguir, apresenta-se uma aplicação que utiliza o serviço de mutex do BRTOS. Neste caso, duas tarefas disputam o uso da USART do Atmega328. Para ter acesso ao recurso, a tarefa de menor prioridade bloqueia o acesso da USART, reservando-a apenas para o seu uso.

Além de se adicionar ao projeto os arquivos **BRTOS.c** e **HAL.c**, é necessário também adicionar os arquivos **serial.c**, **mutex.c** e **queue.c** (das pastas do BRTOS que devem ter sido copiadas para o projeto). Os passos para a execução do programa foram apresentados no exemplo anterior e os arquivos novos do programa são:

1. Mutex_USART_BRTOS.c (programa principal)

```
#include "BRTOS.h"
#include "tasks.h"
#include "serial.h"

#define FOSC 16000000 // Clock Speed
#define BAUD 1200
#define MYBAUD (FOSC/16/BAUD)-1

const CHAR8 Task1Name[] PROGMEM = "Usa_serial_1";
const CHAR8 Task2Name[] PROGMEM = "Usa_serial_2";

PGM_P MainStringTable[] PROGMEM =
{
    Task1Name,
    Task2Name
};

BRTOS_Mutex *TestMutex;
BRTOS_Queue *Serial;

int main(void)
{
    BRTOS_Init();
    Serial_Init(MYBAUD);

    /*Argumentos da função que cria um Mutex: 1 - O endereço de um ponteiro do tipo
     “bloco de controle de mutex” (BRTOS_Mutex) que receberá o endereço do bloco de
     controle alocado para o mutex criado. 2 - A prioridade que será associada ao mutex
     e que deve ser sempre um nível maior do que a maior prioridade das tarefas que
     irão competir por um determinado recurso.3 - A tarefa que recebe o recurso passa a
     prioridade do mutex, não sendo interrompida pelas tarefas que competem pelo mesmo
     recurso. A tarefa volta a sua prioridade original quando liberar este recurso*/

    if (OSMutexCreate(&TestMutex,7) != ALLOC_EVENT_OK)
        while(1){}// Oh Oh - Não deveria entrar aqui !!!

    if(InstallTask(&Usa_serial_1,Task1Name,100,6) != OK)
        while(1){}// Oh Oh - Não deveria entrar aqui !!!

    if(InstallTask(&Usa_serial_2,Task2Name,100,5) != OK)
        while(1){}// Oh Oh - Não deveria entrar aqui !!!
```

```

// Start Task Scheduler
if(BRTOSStart() != OK)
    while(1){}// Oh Oh - Não deveria entrar aqui !!!
while(1);
}
//-----

```

2. tasks.c

```

#include "BRTOS.h"
#include "drivers.h"
#include "tasks.h"

extern BRTOS_Mutex *TestMutex;
extern BRTOS_Queue *Serial;

const CHAR8 SerialTeste1[] PROGMEM = "Esta eh a Tarefa 1";
const CHAR8 SerialTeste2[] PROGMEM = "Agora eh a Tarefa 2";

PGM_P TaskStringTable1[] PROGMEM = { SerialTeste1 };
PGM_P TaskStringTable2[] PROGMEM = { SerialTeste2 };

void Usa_serial_1(void)
{
    while(1)
    {
        strcpy_P(ButtonText, (PGM_P)pgm_read_word(&(TaskStringTable1[0])));
        Serial_Envia_Frase((CHAR8*)ButtonText);
        Serial_Envia_Caracter(LF);
        Serial_Envia_Caracter(CR);
        DelayTask(1);
    }
}
//-----
void Usa_serial_2(void)
{
    while(1)
    {
        // Adquire Mutex
        OSMutexAcquire(TestMutex);
        strcpy_P(ButtonText, (PGM_P)pgm_read_word(&(TaskStringTable2[0])));
        Serial_Envia_Frase((CHAR8*)ButtonText);
        Serial_Envia_Caracter(LF);
        Serial_Envia_Caracter(CR);

        // Libera Mutex
        OSMutexRelease(TestMutex);
        DelayTask(1);
    }
}
//-----

```

3. tasks.h

```
***** OS Tasks *****/
#include "hardware.h"

void Usa_serial_1(void);
void Usa_serial_2(void);

void Transmite_Uptime(void);
void Transmite_Duty_Cycle(void);
void Transmite_RAM_Ocupada(void);
void Transmite_Task_Stacks(void);
void Transmite_CPU_Load(void);
void Reason_of_Reset(void);
```

Exercícios:

21.1 – Pesquise quais são os RTOSs disponíveis atualmente para os sistemas microcontrolados. Por que não se costuma utilizar um RTOS com microcontroladores de 8 bits?

21.2 – Repita o exercício 11.3 (capítulo 11), desta vez empregando o BRTOS.

21.3 – Por que os sistemas embarcados modernos exigem o uso de um RTOS?

22. A IDE DO ARDUINO

A interface de desenvolvimento integrada do Arduino, a chamada IDE ou o programa do Arduino, é apresentada neste capítulo. Ela é simples de usar e permite o desenvolvimento rápido de programas para o ATmega, sem a exigência de profundos conhecimentos técnicos. Permite a prototipação rápida de projetos e tem alcançado cada vez mais usuários.

A IDE apresenta tal grau de abstração, que o usuário não precisa saber o que é um microcontrolador, nem como se utilizam seus registradores de trabalho. Aliado a essa característica, o hardware do Arduino apresenta um suporte crescente na comunidade técnica, com a disponibilidade de bibliotecas de funções para as mais complexas atividades. Tudo associado a um número cada vez maior de módulos prontos para serem conectados ao Arduino (*shields*).

A grande desvantagem de se utilizar a IDE do Arduino é que ela é limitada, não se adequando ao desenvolvimento profissional de projetos. Dentre suas várias limitações não permite depurar nem otimizar o código. Um programa desenvolvido na IDE do Arduino consome mais memória que um programa puramente escrito em C com outra ferramenta de programação, apresentando desempenho inferior.

Assim, a IDE do Arduino é adequada para iniciantes e não é recomendada para profissionais. O suporte para a programação em C para os microcontroladores AVR é ampla e existe uma infinidade de funções prontas para uso, muito superior ao que existe para a IDE do Arduino.

A IDE do Arduino possui uma linguagem própria de programação baseada na linguagem C e C++. Todavia, aceita os comandos e a escrita direta nos registradores do ATmega, da mesma forma como feita no AVR Studio. A IDE pode ser baixada gratuitamente de www.arduino.cc e é executada sem a necessidade da instalação do programa. Na fig. 22.1, é

apresentada a sua interface gráfica com o detalhe dos seus principais comandos (versão 1.0).

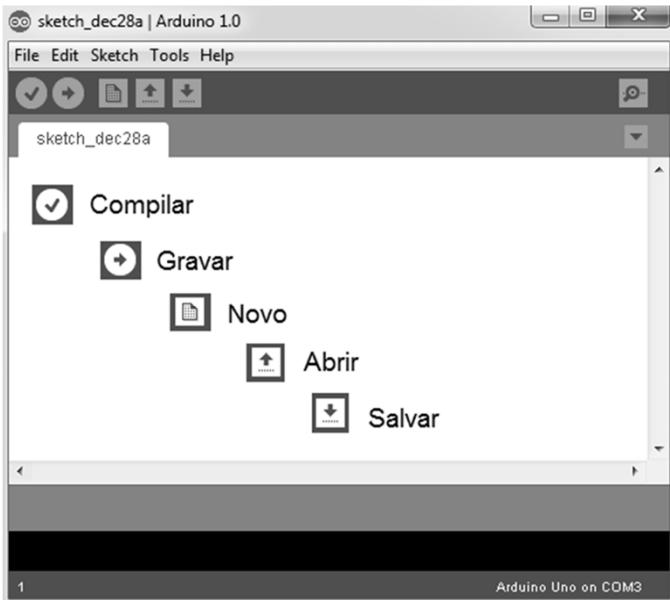


Fig. 22.1 – Programa do Arduino e detalhe dos principais comandos.

O acesso aos principais comandos é direto. O programa pode ser compilado e gravado no Arduino com apenas um clique do *mouse*. Existe também um monitor serial que é uma ferramenta para a comunicação direta com o Arduino através da COM virtual criada na sua instalação.

Após o Arduino ser instalado no computador, será atribuído a ele uma porta COM. Neste momento, é possível gravá-lo. A primeira vez que o programa do Arduino é executado, é necessário dizer qual o modelo do Arduino se está empregando. Isso é feito no menu <Tools><Board>, no caso deste livro, o Arduino Uno. Após a definição de qual modelo de Arduino está conectado ao computador, é necessário definir qual a COM que está associada a ele, isso é feito no menu <Tools><Serial Port>. Para se

descobrir qual COM foi atribuída, pode-se consultar o painel de controle do Windows.

Depois dessas configurações iniciais, o Arduino está pronto para o uso e já pode ser testado com qualquer um dos programas exemplo disponíveis ou com um novo programa. Como existe um LED ligado ao pino PB5 do ATmega328, para teste, geralmente, se grava o programa Blink, disponível no menu <File><Examples><Basics><Blink>. Uma vez que o programa foi selecionado, ele é aberto em uma nova janela.

ACIONANDO UM LED (*Blink*)

Este é o exemplo supracitado, utilizado para piscar um LED no Arduino. Sua funcionalidade é a seguinte: o pino 13 do Arduino é colocado em 5 V, depois de 1 s é colocado em 0 V. Então, depois de 1 segundo o programa volta ao início colocando o pino novamente em 5 V, numa rotina de repetição infinita, resultando no piscar do LED ligado ao pino 13.

```
/*
Blink
Turns on an LED on for one second, then off for one second, repeatedly.
This example code is in the public domain.
*/
void setup() {
    // initialize the digital pin as an output.
    // Pin 13 has an LED connected on most Arduino boards:
    pinMode(13, OUTPUT);
}

void loop() {
    digitalWrite(13, HIGH);    // set the LED on
    delay(1000);              // wait for a second
    digitalWrite(13, LOW);     // set the LED off
    delay(1000);              // wait for a second
}
```

Após o programa ser compilado, na parte inferior da janela de programação, é apresentada a mensagem do compilador indicando o *status* da operação e a quantidade de bytes gerados pelo programa. Depois, basta clicar em gravar (*Upload*) para que o código fonte seja gravado na memória

flash do ATmega328. Se tudo correu bem e nenhuma mensagem de erro foi gerada, o LED ligado ao pino PB5 ficará piscando.

Como observado pelo código exemplo, a programação é simples e direta. A seguir, serão descritas algumas funções e estruturas da programação.

setup()

As inicializações do microcontrolador são feitas na função `setup()`, tal como a definição dos pinos de I/O e outras configurações dos registradores de trabalho empregados.

pinMode(pino, entrada ou saída)

A função `pinMode` define os pinos de entrada e saída, e recebe a definição do pino: um rótulo de 0 a 13, ou A0 a A5, como aparece no *layout* do Arduino. A definição se o pino será entrada ou saída é determinada pelas palavras INPUT ou OUTPUT, respectivamente. No exemplo acima, `pinMode(13,OUTPUT)` indica que o pino 13 do Arduino será um pino de saída, esse pino corresponde ao pino PB5 do ATmega328 (ver a tab. 3.1 do capítulo 3).

loop()

A função `loop()` corresponde ao laço infinito da programação em C. Nela é colocado o código que deve ser continuamente executado.

digitalWrite(pino, estado lógico)

A definição de qual estado lógico deve ser escrito no pino é feita com a função digitalWrite(). Basta indicar qual pino será acionado e qual será o seu estado lógico: HIGH para 5 V e LOW para 0 V.

delay(milissegundos)

A função de atraso delay(ms) corresponde a função _delay_ms() do AVR-GCC. O valor definido nessa função corresponde a quantos milissegundos o programa irá gastar até executar a próxima instrução do código.

LENDO UM BOTÃO (*Button*)

Outro exemplo simples, mostrando como se faz a leitura de um botão, pode ser encontrado no menu <File><Examples><Digital><Button>. Seu código é apresentado a seguir.

```
/*
  Button

  Turns on and off a light emitting diode(LED) connected to digital
  pin 13, when pressing a pushbutton attached to pin 2.

  The circuit:
  * LED attached from pin 13 to ground
  * pushbutton attached to pin 2 from +5V
  * 10K resistor attached to pin 2 from ground
  * Note: on most Arduinos there is already an LED on the board
  attached to pin 13.

  created 2005
  by DojoDave <http://www.0j0.org>
  modified 28 Oct 2010
  by Tom Igoe

  This example code is in the public domain.
  http://www.arduino.cc/en/Tutorial/Button
 */

// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2;      // the number of the pushbutton pin
const int ledPin = 13;        // the number of the LED pin
```

```

// variables will change:
int buttonState = 0;           // variable for reading the pushbutton status

void setup() {
    // initialize the LED pin as an output:
    pinMode(ledPin, OUTPUT);
    // initialize the pushbutton pin as an input:
    pinMode(buttonPin, INPUT);
}

void loop(){
    // read the state of the pushbutton value:
    buttonState = digitalRead(buttonPin);

    // check if the pushbutton is pressed.
    // if it is, the buttonState is HIGH:
    if (buttonState == HIGH) {
        // turn LED on:
        digitalWrite(ledPin, HIGH);
    }
    else {
        // turn LED off:
        digitalWrite(ledPin, LOW);
    }
}

```

digitalRead(pino)

O estado lógico de um pino é lido com o uso desta função. É importante notar que o pino deve estar habilitado como entrada. A função deve passar o estado lógico do botão para uma variável. No programa anterior, o botão estava ligado ao VCC e existia um resistor de *pull-down*. Poderia ter sido empregado o resistor de *pull-up* interno do ATmega328, com um dos terminais do botão ligado ao terra (ver o capítulo 5).

As instruções <const int buttonPin 2> e <const int ledPin 13> definem um nome para os pinos do botão e do LED para facilitar a escrita do programa. No laço infinito, se o botão for pressionado <if(buttonState==HIGH)> o led é ligado, caso contrário desligado. A variável <int buttonState> serve para armazenar o estado lógico do botão.

São inúmeras as funções e detalhes da linguagem utilizada na IDE do Arduino. No menu <Help><Reference> todos esses detalhes podem ser encontrados com explicações detalhadas do seu funcionamento. Um

programa pode ser dividido em 3 partes: estrutura, valores (variáveis e constantes) e funções, resumidas na tab. 22.1.

Tab. 22.1 – Estruturas, valores e funções para a programação na IDE do Arduino.

ESTRUTURAS	VARIÁVEIS	FUNÇÕES
setup() loop()	Constantes HIGH LOW INPUT OUTPUT true false integer constants floating point constants	I/O digital pinMode() digitalWrite() digitalRead()
Estruturas de controle if if...else for switch case while do... while break continue return goto	Tipos de dados void boolean char unsigned char byte int unsigned int word long unsigned long float double string - char array String - object array	I/O analógico analogReference() analogRead() analogWrite() - PWM
Outras sintaxes ; (ponto-e-vírgula) { } (chaves) // (comentário de linha simples) /* */ (comentário de várias linhas) #define #include		I/O avançado tone() noTone() shiftOut() pulseIn()
Operadores aritméticos = (atribuição) + (soma) - (subtração) * (multiplicação) / (divisão) % (módulo)	Conversão char() byte() int() word() long() float()	Tempo millis() Matemáticas min() max() abs() constrain() map() pow() sqrt()
Operadores de comparação == (igual) != (diferente) < (menor que) > (maior que) <= (menor ou igual) >= (maior ou igual)	Variáveis de escopo e qualificadores variable scope static volatile const	Trigonometria sin() cos() tan()
Operadores lógicos && (e) (ou) ! (não)	Utilidades sizeof()	Números aleatórios randomSeed() random()

Operadores para ponteiros	Bits e bytes
* valor/declaração	lowByte()
& endereço	highByte()
Operadores bit a bit	bitRead()
& (E)	bitWrite()
(OU)	bitSet()
^ (OU EXCLUSIVO)	bitClear()
~ (NOT)	bit()
<< (desloca p/ esquerda)	Interrupções externas
>> (desloca p/ direita)	attachInterrupt()
Operadores de composição	detachInterrupt()
++ (incremento)	Interrupções
-- (decremento)	interrupts()
+= (adição)	noInterrupts()
-= (subtração)	Comunicação
*= (multiplicação)	Serial
/= (divisão)	
&= (E bit a bit)	
= (OU bit a bit)	

A estrutura da programação é muito similar a utilizada na linguagem C. Com o uso e familiaridade, a programação vai se tornando mais fácil e rápida. Os inúmeros exemplos disponíveis no menu <File><Examples>, em conjunto com o <Help>, ajudam na compreensão das funções.

Apesar de ser bastante limitada, a IDE do Arduino é útil quando se deseja testar alguma ideia em um projeto preliminar, onde o tempo é crucial e as limitações do Arduino não são um problema.

23. FUSÍVEIS E A GRAVAÇÃO DO ATMEGA328

Neste capítulo, são explicitados os detalhes para a gravação das memórias *flash* e EEPROM do ATmega e a configuração dos fusíveis de gravação e bloqueio, os quais são utilizados para selecionar as características desejadas de trabalho do microcontrolador. Explica-se o uso do *Watchdog Timer* e como deve ser o hardware para permitir a gravação *In-system* através da SPI.

Para que o microcontrolador possa executar o programa desenvolvido, depois deste ter sido compilado e gerado o código de máquina, é necessário gravar esse código na memória *flash* do microcontrolador. Neste momento, as características que geralmente não são definidas pelo código de programação estarão disponíveis via interface de gravação. Dentre as principais definições que podem ser ajustadas estão:

- Fonte de *clock* empregada (externa, interna, cristal, frequência, etc.).
- Ajuste do *Power-on-Reset* (tempo de inicialização automática).
- Ajuste do *Brown-out* (tensão mínima de trabalho para inicialização automática).
- Habilitação do *debug Wire* (sistema para *debug In-System*).
- Habilitação do *Watchdog Timer* (temporizador de segurança contra o travamento do programa).
- Gravação da EEPROM com valores definidos previamente.
- Proteção do código contra cópia não autorizada.
- Emprego de *boot loader* (código para autogravação).
- Habilitação da gravação serial, via SPI.

A fonte de *clock* irá definir que tipo de oscilador a CPU irá empregar e qual tipo de componente eletrônico está sendo usado para esse fim. É possível, por exemplo, definir o uso da fonte interna sem o uso de componentes externos, bem como a sua frequência.

O *Power-on Reset* é empregado para a definição do tempo que a CPU levará para se auto inicializar após a energização do circuito. Garante um *reset* preciso na energização; com isso, o emprego de uma rede RC para inicializar fisicamente o microcontrolador é desnecessária e o pino de *reset* pode ser ligado diretamente ao VCC, ou no caso do ATmega328, ser utilizado como pino de I/O.

O *Brown-out* inicializa a CPU caso a tensão de energização caia abaixo de um valor escolhido. Dessa forma, garante o correto funcionamento do software no caso de instabilidade na alimentação.

O *Watchdog Timer* é fundamental para evitar o travamento do software, principalmente em aplicações críticas.

O *debug Wire* é um sistema que permite a depuração do programa no chip, usa uma única via de comunicação (pino de *Reset*) e permite o controle do fluxo do programa diretamente no microcontrolador.

É possível gravar a EEPROM com valores pré-definidos, armazenando, por exemplo, tabelas. Além disso, empregando os *lock* bits é possível garantir que o código não seja copiado.

23.1 OS FUSÍVEIS E BITS DE BLOQUEIO

O ATmega328 possui 4 bytes para configuração. As características de funcionamento são definidas em 3 bytes: byte baixo (LOW), byte alto (HIGH) e byte estendido (EXTENDED). Os bits desses bytes são chamados de fusíveis, pois ligam ou desligam a característica desejada. A proteção de memória é feita em um byte com os chamados bits de bloqueio (LOCK) de segurança. Essa característica permite proteger o código contra cópias. Na tab. 23.1, são apresentados esses bytes e seus respectivos bits.

Tab. 23.1 – Bytes de configuração do ATmega328.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
EXTENDED	-	-	-	-	-	BODLEVEL2	BODLEVEL1	BODLEVEL0
HIGH	RSTDISBL	DWEN	SPIEN	WDTON	EESAVE	BOOTSZ1	BOOTSZ0	BOOTRST
LOW	CKDIV8	CKOUT	SUT1	SUTO	CKSEL3	CKSEL2	CKSEL1	CKSEL0
LOCK BITS	-	-	BLB12	BLB11	BLB02	BLB01	BL2	BL1

Quando o ATmega for gravado, os programas de gravação permitem a escrita direta dos bytes de configuração ou a seleção individual dos bits. Por exemplo, para o Arduino com o ATmega328, esses valores são:

EXTENDED BYTE	=	0x05 (no AVR Studio é 0xFD)
HIGH BYTE	=	0xDE
LOW BYTE	=	0xFF
LOCK BITS	=	0xCF

Os bits de configuração são detalhados nas tabs. 23.2-5. Estes detalhes podem parecer, à primeira vista, um pouco confusos. Todavia, os programas de gravação possuem uma interface amigável, o que facilita muito a escolha das características desejadas, como nos casos do AVR Studio ou do AVR-Burn-O-Mat. O sítio www.engbedded.com/fusecalc/ também permite a fácil determinação dos valores hexadecimais dos fusíveis de programação.

Tab. 23.2 – Fusíveis de programação do byte de configuração EXTENDED.

Bits do Byte EXTENDED	Nr. Bit	Descrição	Valor Default
BODLEVEL2	2	Seleciona o nível de tensão para reinicialização caso a tensão de alimentação caia abaixo do valor pré-determinado:	1
BODLEVEL1	1		1
BODLEVEL0	0	111 – desligado 110 – 1,8 V 101 – 2,7 V 100 – 4,3 V	1

Tab. 23.3 – Fusíveis de programação do byte de configuração HIGH.

Bits do Byte HIGH	Nr. Bit	Descrição	Valor Default
RSTDISBL	7	Seleção do pino PC6 como <i>Reset</i> ou I/O	1 (não programado, PC6 é o pino de <i>Reset</i>)
DWEN	6	Habilita a depuração por <i>debug Wire</i>	1 (não programado)
SPIEN	5	<u>Habilita a programação serial</u>	0 (programado, SPI habilitada)
WDTON	4	<i>Watchdog</i> sempre ligado	1 (não programado)
EESAVE	3	A memória EEPROM é preservada quando o chip for apagado	1 (não programado, EEPROM não preservada)
BOOTSZ1	2	Define o tamanho da região de <i>boot loader</i> : 00 – 2048 words, início no endereço 0x3800 01 – 1024 words, início no endereço 0x3C00	0
BOOTSZ0	1	10 – 512 words, início no endereço 0x3E00 11 – 256 words, início no endereço 0x3F00	0
BOOTRST	0	Define o endereço de início de execução do programa após a inicialização: 0 – endereço do <i>boot loader</i> 1 – início da <i>flash</i> (0x0000)	1

Tab. 23.4 – Fusíveis de programação do byte de configuração LOW.

Bits do Byte LOW	Nr. Bit	Descrição	Valor Default
CKDIV8	7	Divide o <i>clock</i> de trabalho por 8	0 (programado)
CKOUT	6	Habilita o sinal de <i>clock</i> no pino CLKO	1 (não programado)
SUT1	5	Em conjunto com os bits CKSELx define o tempo para a inicialização.	1
SUTO	4		0
CKSEL3	3	Seleção da fonte de <i>clock</i> :	0
CKSEL2	2	0000 – sinal externo	0
CKSEL1	1	0010 – fonte interna (RC) de 8 MHz	1
CKSEL0	0	0011 – fonte interna (RC) de 128 kHz 010x – cristal externo de 32 kHz 011x – cristal ou ressonador externo (<i>full swing</i>) 1xxx – cristal ou ressonador externo (<i>low power</i>)	0

Tab. 23.5 – Bits para bloqueio da memória.

Bit	Nr. Bit	Descrição	Valor Default
BLB12	5	Controlam o acesso a região de aplicação (software) da <i>flash</i> através das instruções LPM e SPM: 00: a instrução SPM não pode gravar na área de aplicação. LPM executado no <i>boot loader</i> não pode ler a área de aplicação. 01: LPM executada no <i>boot loader</i> não pode ler a área da aplicação. 10: a instrução SPM não pode gravar na área de aplicação. 11: sem restrições.	1
BLB11	4		1
BLB02	3	Controlam o acesso a região do <i>boot loader</i> através das instruções LPM e SPM: 00: a instrução SPM não pode gravar na área de <i>boot loader</i> . LPM executado na aplicação não pode ler a área de <i>boot loader</i> . 01: LPM executada na aplicação não pode ler a área do <i>boot loader</i> . 10: a instrução SPM não pode gravar na área de <i>boot loader</i> . 11: sem restrições.	1
BLB01	2		1
LB2	1	11 – acesso livre a <i>flash</i> , fusíveis e <i>lock bits</i> . 10 – acesso bloqueado à <i>flash</i> e aos fusíveis.	1
LB1	0	00 – acesso bloqueado à <i>flash</i> , fusíveis e <i>lock bits</i> .	1

Se a programação SPI pretende ser empregada, é necessário ter muito cuidado ao se gravar os bits RSTDISBL e SPIEN (byte HIGH). Se o pino PC6 (*reset*) do ATmega328 for utilizado como I/O, a gravação SPI não poderá mais ser realizada. Quando o microcontrolador estiver no circuito, ele será gravado via SPI. Desabilitá-la accidentalmente, impedirá a gravação do componente no circuito (o que é impraticável para componentes SMD). Da mesma forma, é preciso ter cuidado com a seleção da fonte de *clock*; se for escolhida uma fonte inexistente como, por exemplo, um cristal externo, quando o circuito do microcontrolador não o dispõe, a gravação será impossibilitada porque não haverá fonte de *clock* para o trabalho do microcontrolador. A gravação só poderá ser realizada novamente após o suprimento dessa fonte.

Caso os bits de travamento da memória estejam ativos, eles só podem ser alterados com o apagamento completo do chip.

23.2 HARDWARE DE GRAVAÇÃO

A gravação do ATmega necessita de um hardware específico. No caso do Arduino, sua plataforma já dispõe desse hardware, bastando o software de controle. Se for utilizada a IDE do Arduino, o controle dos fusíveis de programação é transparente ao usuário, visto que não precisam ser alterados. Todavia, o projetista, cedo ou tarde, vai se deparar com a necessidade de gravar um microcontrolador fora da plataforma Arduino. Isso até pode ser feito com o Arduino (ver fórum do Arduino - www.arduino.cc). Entretanto, o hardware do Arduino apresenta limitações e não pode ser utilizado simplesmente substituindo-se o ATmega328 existente por um que se deseja gravar.

O ATmega328 do Arduino utiliza um programa de *boot loader* para poder gerenciar sua própria gravação, o qual vem gravado no Arduino. Isso é uma vantagem porque simplifica o hardware. Então, basta uma simples interface de comunicação serial com o computador, pois a gravação é feita através da USART (pinos TX e RX do microcontrolador).

Porém, utilizar um *boot loader* consome memória do microcontrolador e, devido as taxas de comunicação da USART, não permite gravações rápidas, o que é um problema quando se necessita gravar microcontroladores com programas extensos em escala comercial. Também, pode existir o problema de auto inicialização: o *boot loader* sempre é inicializado quando é aberta a comunicação serial com o microcontrolador. Isso pode produzir o travamento do microcontrolador caso o ATmega328 receba informações que não sejam para a gravação antes do *boot loader* passar o controle ao programa principal.

Outro problema ocorre se o ATmega328 do Arduino precisar ser substituído. Seria necessário gravar o *boot loader* no novo ATmega328 e não existe forma de fazer isso sem um gravador externo.

Existem inúmeros gravadores disponíveis no mercado. A Atmel produz o AVR *Dragon*, um gravador profissional que permite várias formas de

gravação, depuração e, ainda, a emulação de alguns microcontroladores. Entretanto, existem inúmeros gravadores baratos e de confecção simples, com código, desenho da PCI e circuito disponíveis gratuitamente na internet. Esse é o caso do USBTiny, um dos mais populares. Portanto, é fácil obter um gravador para os microcontroladores AVR.

O AVR *Studio* será utilizado para ilustrar o funcionamento de um gravador. Dependendo do tipo de gravador, o AVR *Studio* não dará suporte à gravação. Neste caso, pode ser empregado outro programa, como por exemplo, o AVR Burn-O-Mat (ou o AVRdude com outra interface gráfica) ou o PonyProg.

Para acessar a ferramenta de gravação é necessário acessar o menu <Tools> <AVR Programming>, fig. 23.1. Será aberta a janela da fig. 23.2, na qual se pode escolher o hardware de gravação, que deve estar conectado ao computador. É possível, também, selecionar o <AVR Simulator> e ter acesso a todas as funcionalidades da janela de gravação sem dispor do hardware. Isso é interessante para se determinar o valor dos bytes de configuração, que podem ser utilizados em outro programa.

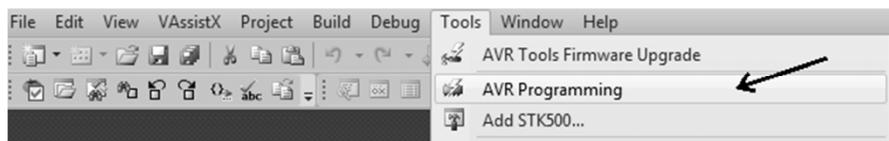


Fig. 23.1 – Acessando a ferramenta de gravação do AVR Studio.

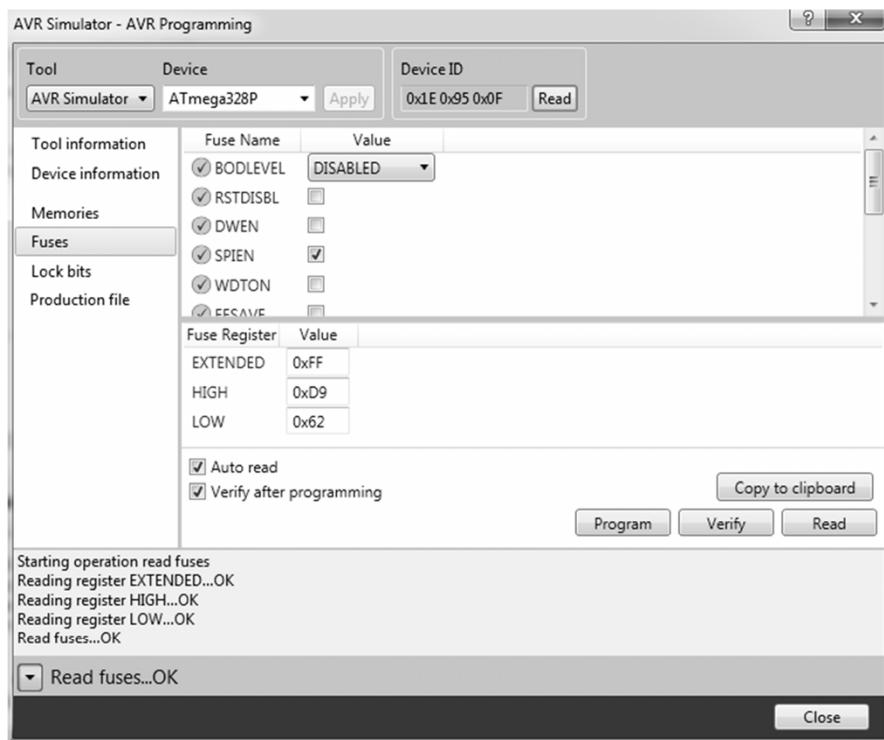


Fig. 23.2 – Janela do AVR Studio para a gravação da memória do microcontrolador.

Na janela de gravação também são especificados os arquivos para a gravação da memória de programa (*.hex) e para a memória EEPROM (*.eep), bem como o tipo de microcontrolador para a gravação, o qual pode ter sua identificação conferida na seção <Device ID>. Por exemplo, o ATmega328P apresenta o valor: 0x1E 0x95 0x0F.

Além dos arquivo *.hex, com o código do programa, e do arquivo *.eep, com os dados para a EEPROM, é possível utilizar um único arquivo com todos os dados a serem gravados no microcontrolador, incluindo os fusíveis e os bits de bloqueio. Esse arquivo possui extensão *.elf (*Executable and Linkable Format*) e é muito útil quando o microcontrolador deve ser gravado em uma linha de produção, pois a gravação com um único arquivo

aumenta a velocidade de gravação e diminui a probabilidade de erros humanos no processo.

O compilador GCC gera automaticamente o arquivo *.elf. Todavia, a definição dos dados para a EEPROM e a definição dos bytes de configuração devem ser feitos através da programação (no capítulo 7, mostrou-se como gerar os dados para a EEPROM). Para o ajuste dos bytes de configuração, é necessário incluir no programa duas novas bibliotecas, como exemplificado abaixo.

```
//-----
#include <avr/fuse.h>
#include <avr/lock.h>

//gerando os dados para gravar o valor dos fusíveis no arquivo *.elf
FUSES =
{
    .low = 0xFF,
    .high = 0xDA,
    .extended = 0xFD,
};

//gerando os dados para gravar os bits de travamento no arquivo *.elf
LOCKBITS = 0xCF;

int main()
{}
```

Após a gravação, raramente o programa gravado não necessitará alterações para satisfazer as características desejadas de trabalho. Logo, é bom ter ciência que a memória *flash* será gravada inúmeras vezes até se chegar a uma versão satisfatória do programa.

23.3 O TEMPORIZADOR WATCHDOG

Por mais que o programador teste seu programa, somente a prática vai definir o sucesso do programa. Programas complexos e até mesmo programas simples podem ter sua estabilidade comprometida por certos fatores imprevisíveis durante a depuração e testes. Esse é um fator comum quando se utilizam computadores. Quem já não teve algum software de computador travado em alguma ocasião?

Em sistemas microcontrolados e, principalmente, em sistemas de tempo real onde a resposta do sistema possui um tempo definido, é fundamental que qualquer eventual travamento do sistema seja corrigido rapidamente.

Em sistemas microcontrolados que lidam com a vida ou que não podem falhar porque produzirão grandes danos, em caso de travamento, o sistema deve voltar o mais rápido possível à atividade normal, ou mesmo, após sucessivas tentativas, acionar um mecanismo de segurança. Exemplos de tais sistemas são: equipamentos médicos, sistemas de controle em aviões e carros, sistemas de controle em indústrias, semáforos de trânsito, usinas geradoras de energia elétrica, sistemas militares e sistemas espaciais.

Para impedir o travamento do programa em microcontroladores, emprega-se o temporizador *Watchdog*. Ele é um temporizador que se habilitado, deve ser inicializado pelo programa periodicamente; caso isso não ocorra e o temporizador estoure sua contagem, o microcontrolador é automaticamente inicializado.

O ATmega328 possui um temporizador *Watchdog* (WDT) que utiliza um oscilador, separado no chip, de 128 kHz. O WDT pode inicializar o sistema ou, ainda, gerar um pedido de interrupção. No modo normal de operação, é necessário usar a instrução de inicialização do contador (WDR - *Watchdog Timer Reset*) para zerá-lo antes que ele atinja o valor de estouro; caso contrário, a inicialização do microcontrolador será realizada.

No modo de interrupção, o WDT produz uma interrupção quando o tempo limite de estouro é alcançado. Essa interrupção pode ser utilizada para ‘despertar’ o dispositivo de algum modo de economia de energia (*sleep modes*) ou como uma temporização geral para o sistema. Por exemplo, ele pode ser empregado para limitar o tempo máximo permitido para certa operação, gerando uma interrupção quando a operação demorar mais que o esperado.

No modo de inicialização do sistema, o WDT irá inicializar o microcontrolador quando o tempo limite for alcançado, impedindo o travamento do programa.

Existe ainda um terceiro modo de operação, que é a combinação dos dois anteriores, primeiro gerando uma interrupção e, então, uma inicialização do sistema. Isso permite, por exemplo, o salvamento de parâmetros importantes antes da inicialização.

Se o fusível (WDTON) de habilitação do WDT for programado, o ATmega estará no modo de inicialização. Então, o bit desse modo (WDE) e o de interrupção (WDIE) serão mantidos em 1 e 0, respectivamente. Para garantir o funcionamento seguro do WDT, a sua configuração deve seguir a sequência de escrita no registrador WDTCSR:

1. Na mesma operação, escrever 1 lógico no bit WDCE e WDE. A escrita em WDE deve ser feita independente do seu valor prévio.
2. Nos próximos 4 ciclos de *clock*, escrever 1 lógico em WDE e configurar os bits de *prescaler* (WDP) como desejado, agora com o bit WDCE em zero. Isso deve ser feito em uma única operação.

A seguir são apresentadas duas funções, uma para desligar o WDT e outra para alterar o seu valor de prescaler, conforme o manual do ATmega328.

```
-----  
#include <avr/io.h>  
#include <avr/interrupt.h>  
#include <avr/wdt.h>  
  
void WDT_off(void)  
{  
    cli();                                //desabilita as interrupções  
    wdt_reset();  
  
    MCUSR &= ~(1<<WDRF);           //limpa o flag sinalizador do WDT  
    WDTCSR |= (1<<WDCE) | (1<<WDE); /*escreve 1 em WDCE e WDE. Mantém valores prévios  
                                         do prescaler para prevenir um estouro não intencional*/  
    WDTCSR = 0x00;                      //desliga o WDT  
    sei();                                //habilita as interrupções  
}  
-----
```

```

void WDT_Prescaler_Change(void)
{
    cli();      //desabilita interrupções

    wdt_reset();
    WDTCSR |= (1<<WDCE) | (1<<WDE);      //sequência de inicialização
    WDTCSR = (1<<WDE) | (1<<WDP2) | (1<<WDP0);/*ajusta o novo valor de prescaler
                                                (tempo de estouro)*/
    sei();      //habilita interrupções
}
//-----

```

Nota: se o WDT é acidentalmente habilitado, como por exemplo, com um ponteiro perdido ou uma condição de *brown-out*, o microcontrolador será inicializado e o WDT permanecerá habilitado. Se o código não lidar com o WDT, isso gerará um laço infinito de inicializações. Para evitar essa situação, o programa de aplicação deve sempre limpar o bit de sinalização WDRF e o bit de controle WDE na rotina de inicialização do microcontrolador, mesmo se o WDT não for empregado⁵⁵.

DESCRIÇÃO DOS REGISTRADORES

MCUSR – MCU Status Register

Bit	7	6	5	4	3	2	1	0
MCUSR	-	-	-	-	WDRF	BORF	EXTRF	PORF
Lê/Escrive	L	L	L	L	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	-	-	-	-

Bit 3 – WDRF – Watchdog System Reset Flag

Este bit é colocado em 1 se uma inicialização do WDT acontecer. Este bit é limpo pelo *Power-on-Reset* ou pela escrita de zero.

Bit 2 – BORF – Brown-out Reset Flag

Colocado em 1 se um *brown-out* ocorrer. Este bit é limpo pelo *Power-on-Reset* ou pela escrita de zero.

Bit 1 – EXTRF – External Reset Flag

Colocado em 1 se uma inicialização externa ocorrer. Este bit é limpo pelo *Power-on-Reset* ou pela escrita de zero.

Bit 0 – PORF – Power-on Reset Flag

Colocado em 1 se um *Power-on* ocorrer. Este bit é limpo somente pela escrita de zero.

⁵⁵ Tal procedimento não foi adotado em nenhum programa deste livro.

WDTCSR – Watchdog Timer Control Register

Bit	7	6	5	4	3	2	1	0
	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E

Valor Inicial

0	0	0	0	-	0	0	0
---	---	---	---	---	---	---	---

Bit 7 – WDIF – Watchdog Interrupt Flag

Este bit é colocado em 1 se um estouro do WDT acontecer e sua interrupção estiver habilitada. É limpo por hardware quando a rotina de interrupção é executada.

Bit 6 – WDIE – Watchdog Interrupt Enable

Quando este bit estiver em 1, bem como o bit I do registrador SREG, a interrupção do WDT estará habilitada. Se WDE = 0 e WDIE = 1, o WDT estará no modo de interrupção e uma interrupção será executada cada vez que o contador estourar. Se WDE = 1 e WDIE = 1, o WDT estará no modo de interrupção e de inicialização. No primeiro estouro do WDT, WDIF será colocado em 1, a execução da interrupção irá limpar automaticamente WDIE e WDIF e o WDT irá para o modo de inicialização. Para permanecer no modo de interrupção e de inicialização, WDIE deve ser colocado em 1 a cada interrupção (o que não deve ser feito dentro da rotina de interrupção). Na tab. 23.6, são apresentados os modos de configuração para o WDT.

Tab. 23.6 – Configurações para os modos de operação do WDT.

WDTON	WDE	WDIE	Modo	Ação no estouro
1	0	0	Parado	Nenhuma
1	0	1	Interrupção	Desvio para o vetor de interrupção
1	1	0	Inicialização do sistema	Inicialização
1	1	1	Interrupção e Inicialização do sistema.	Desvio para o vetor de interrupção, então vai para o modo de inicialização do sistema.
0	x	x	Inicialização do sistema	Inicialização

* O fusível WDTON em zero significa programado.

Bit 4 – WDCE – Watchdog Change Enable

Este bit é utilizado em sequências de temporização para mudar WDE e os bits de *prescaler*. Para limpar WDE e/ou mudar o *prescaler*, WDCE deve ser 1. Uma vez colocado em 1, o hardware o limpará em 4 ciclos de *clock*.

Bit 3 – WDE – Watchdog System Reset Enable

WDE é sobreescrito por WDRF no MCUSR. Isso significa que WDE é sempre 1 quando WDRF é 1. Para limpar WDE, WDRF deve ser limpo primeiro.

Bits 5, 2:0 – WDP3:0 – Watchdog Timer Prescaler 3, 2, 1 e 0

Estes bits determinam o valor do *prescaler* para o WDT e, portanto, o seu tempo limite, conforme tab. 23.7.

Tab. 23.7 – Seleção dos valores de prescaler para o WDT.

WDP3	WDP2	WDP1	WDP0	Nr. de ciclos do oscilador do WDT	Tempo limite típico com VCC = 5V
0	0	0	0	2k -2048	16 ms
0	0	0	1	4k - 4096	32 ms
0	0	1	0	8k - 8192	64 ms
0	0	1	1	16k - 16384	0,125 s
0	1	0	0	32k - 32768	0,25 s
0	1	0	1	64k - 65536	0,5 s
0	1	1	0	128 k - 131072	1,0 s
0	1	1	1	256 k - 262144	2,0 s
1	0	0	0	512 k - 524288	4,0 s
1	0	0	1	1024 k - 1048576	8,0 s
-	-	-	-	demais valores reservados	-

Exercício:

23.1 – Empregando o WDT do ATmega328 faça um programa utilizando o modo de interrupção para piscar um LED a cada 0,5 s.

23.4 GRAVAÇÃO IN-SYSTEM DO ATMEGA

Quando se faz um projeto microcontrolado, ou o microcontrolador é soldado à placa de circuito impresso (PCI), ou se utiliza algum soquete que permita a sua substituição. A solda é a melhor solução dado a eliminação do soquete e, quando se utilizam componentes SMD, resulta na diminuição do tamanho físico da PCI. É, portanto, a solução mais econômica e moderna.

Quando o microcontrolador pode ser retirado facilmente da PCI, a gravação de sua memória pode ser externa. Porém, quando o mesmo é soldado ou não se quer retirá-lo do circuito, é necessário gravar o microcontrolador na própria PCI, a chamada programação *In-System* (ISP).

No ATmega é possível utilizar a interface SPI para a programação da sua memória e dos seus bytes de configuração. A SPI permite uma gravação eficiente, rápida e com o emprego de hardwares de gravação baratos e simples, pois exige poucos componentes eletrônicos.

Para a ISP⁵⁶, a Atmel recomenda a configuração de conector mostrado na fig. 23.3 (conector em barra duplo – CONN-DIL6). Na tab. 23.8, é apresentada a funcionalidade dos seus pinos.

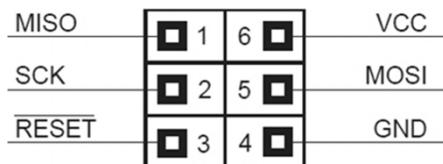


Fig. 23.3 – *Layout* do conector para gravação *In-System*, vista superior (CONN-DIL6).

Tab. 23.8 – Descrição dos pinos para a ISP.

Pino	Nome	Comentário
SCK	Clock serial	<i>Clock</i> de programação gerado pelo gravador (mestre).
MOSI	Saída do mestre entrada do escravo	Linha de comunicação do mestre (gravador) para o escravo (microcontrolador a ser gravado).
MISO	Entrada do mestre saída do escravo	Linha de comunicação do escravo (microcontrolador a ser gravado) para o mestre (gravador).
GND	Terra comum	Mestre e escravo devem compartilhar o mesmo terra.
RESET	<i>Reset</i> do microcontrolador	Para habilitar a gravação <i>In-System</i> , o <i>reset</i> do microcontrolador (escravo) deve ser mantido ativo pelo gravador.
VCC	Alimentação do microcontrolador	O gravador pode fornecer alimentação ao microcontrolador ou utilizar a alimentação deste na programação. Assim, o microcontrolador pode operar em qualquer tensão.

Cuidados devem ser tomados quando os pinos do microcontrolador são empregados na programação *In-System* e, também, pelo hardware em outras funções. Deve-se evitar que essas funções absorvam corrente do sistema de gravação. Um exemplo da conexão do ATmega328 para a ISP é apresentada na fig. 23.4. Nesse caso, se perde o pino de *reset* como possível I/O. O circuito pode ser o mesmo para diferentes AVR's.

⁵⁶ Ver o *application note* da Atmel: *In-System Programming*.

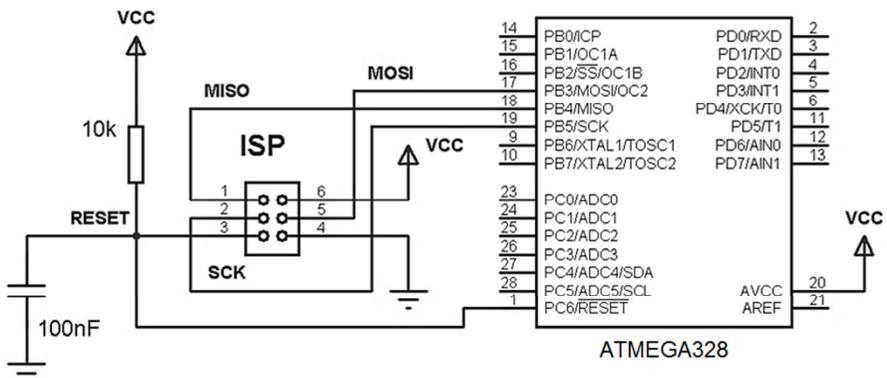


Fig. 23.4 – Conexão do ATmega328 para a ISP.

Exercício:

23.2 – Consulte o manual do ATmega328 e descubra como funciona o *DebugWire*.

24. CONSIDERAÇÕES FINAIS

O projeto de sistemas microcontrolados exige muito estudo, dedicação e experiência. Cada projeto irá exigir conhecimentos específicos que podem demandar muita pesquisa. As informações mais completas sobre determinado microcontrolador são encontradas nos manuais fornecidos pelos fabricantes. Os livros sobre microcontroladores específicos são baseados nesses manuais, associados também à experiência do escritor e, geralmente, contém dicas importantes sobre técnicas de hardware e software.

É fundamental ao projetista conhecer as partes que compõe o projeto para produzir um programa eficiente e de forma rápida. Mesmo quando um projeto é realizado em equipe, as atividades divididas são integradas com a troca de informações entre os membros constituintes do projeto.

Quando se realiza um projeto, deve-se considerar, principalmente, o custo da execução do mesmo. As seguintes questões ditarão o andamento do projeto:

- Quais são os componentes eletrônicos mais baratos que podem ser utilizados no circuito e qual a disponibilidade deles no mercado?
- Quanto tempo é exigido para a confecção do projeto?
- Qual a experiência e os conhecimentos dos projetistas envolvidos?
- Quais as normas de compatibilidade eletromagnética que devem ser respeitadas?
- O circuito será alimentado por baterias?
- Onde será instalada a placa de circuito impresso? Qual será o invólucro dela e que tipo de desenho deve apresentar?

É importante lembrar que um bom projeto implica em um código eficiente e no emprego do menor número de componentes possíveis, mantendo a qualidade do resultado desejado. Vários anos de experiência são necessários para se formar um bom projetista e programador. Além disso, de nada adianta bons programa e circuito, se o projeto da placa de circuito impresso não for bem feito, impedindo o adequado funcionamento do circuito e, ainda, implicando em manutenções constantes.

Dada a portabilidade do código C, bem como a necessidade de não se reescrever códigos, é altamente recomendado a estruturação do projeto em arquivos separados de acordo com a funcionalidade, permitindo a criação de bibliotecas próprias. No estudo de rotinas prontas, é fundamental a completa compreensão dessas, caso contrário, a habilidade de programação não se desenvolve.

É importante para o programador desenvolver seus algoritmos previamente à programação. Só programadores muito experientes chegam a desprezar esse fato. O erro mais comum de quem está aprendendo a programar é sentar em frente à tela do computador e começar a digitar linhas de código sem uma boa consideração de como realizar determinada atividade, a chamada programação ‘Darwiniana’, tentativas e erros que devem levar o programa a ‘evoluir’. Isso acarreta programas extensos, inefficientes, difíceis de corrigir e que demandam um tempo longo de programação. O tempo despendido no desenvolvimento de um algoritmo produzirá eficiência exponencial na programação. Assim, um bom programador necessita de pelo menos lápis e papel para rascunhar o fluxograma de trabalho do seu programa.

Na área de microcontroladores, atualizar-se é questão de sobrevivência. Não se deve ficar preso a uma única tecnologia. O mercado muda constantemente e sempre serão os requisitos do projeto que indicarão qual a melhor tecnologia a ser empregada.

Todo estudante e muitos programadores de microcontroladores se perguntam: qual é o melhor microcontrolador? A resposta depende de

inúmeros fatores, entre eles, a complexidade do projeto. Se o projeto for complexo e demandar programas que exijam inúmeros cálculos e precisem de tempos de resposta rápidos, é bom começar a pensar em arquiteturas de 32 bits. Nesse ponto, na atualidade, microcontroladores baseados nos núcleos ARM (*Advanced RISC Machine*) dominam o mercado. Se o projeto exigir baterias, um dos microcontroladores mais eficientes nesse quesito é o MSP430 da Texas.

Mesmo que se chegue à conclusão que o microcontrolador A é melhor que o B, o custo-benefício ainda será mandatório. Se o programador não conhece o microcontrolador considerado ideal, o tempo que ele vai gastar para estudar e compreender um novo microcontrolador pode gerar um atraso ou um custo inaceitável ao projeto. Entretanto, de acordo com os requisitos exigidos, a substituição do microcontrolador pode ser imprescindível. O gasto inicial na sua substituição pode ser compensado com um ganho de desempenho, incluindo a diminuição do números de componentes eletrônicos do projeto.

Muitas pessoas afirmam que o melhor microcontrolador é aquele que você domina e, se você domina bem um determinado microcontrolador, a migração para outra tecnologia será rápida, pois algoritmos não mudam!

Com o crescente avanço da eletrônica, o número de interfaces de comunicação entre os microcontroladores e o mundo externo tem aumentado exponencialmente. No caso específico do ATmega328, elas são três: SPI, USART e TWI. Existem inúmeras outras, como: a USB (*Universal Serial Bus*), cada vez mais integrada aos microcontroladores; a CAN (*Control Area Network*), muito empregada em ‘chão de fábrica’ e em automóveis; a Ethernet, onde sistemas microcontrolados são ligados diretamente a *World Wide Web*. Muitas vezes, a escolha do microcontrolador vai ser feita devido a existência de um determinado periférico.

As comunicações sem fio também têm crescido, os padrões ZigBee e *Bluetooth* têm aparecido em um número cada vez maior de dispositivos

eletrônicos. Alguns microcontroladores já possuem periféricos para esse tipo de comunicação.

Com a passagem do tempo, módulos prontos estão sendo confeccionados para serem empregados diretamente com sistemas microcontrolados, como por exemplo: módulos GPS, módulos Ethernet, módulos ZigBee, módulos *Bluetooth*, módulos para cartões SD, módulos LCD e módulos para o acionamento de motores. A tendência é a produção de módulos com cada vez mais funcionalidades. O objetivo dos fabricantes é oferecer ferramentas para o desenvolvimento rápido de projetos, permitindo ao projetista gastar pouco tempo no desenvolvimento do software e hardware, simplificando o projeto e diminuindo os seus custos.

Apesar de microcontroladores de 8 bits possuírem relativamente pouca memória e baixo desempenho de processamento, é possível utilizá-los com um sistemas operacional de tempo real embarcado (RTOS). Isso ocorre porque atualmente esses microcontroladores já dispõem de memória capaz de suportar RTOS mínimos e, a cada dia, estão surgindo novos RTOS capazes de rodar nessas arquiteturas. O emprego de um RTOS em um microcontrolador de 8 bits deve ser avaliado com cuidado, dado o seu impacto no desempenho e consumo de memória. Todavia, um RTOS é uma ferramenta poderosa para a resolução de problemas, difíceis de solucionar com a programação tradicional. Logo, vale a pena aprender a usá-los.

Os RTOS são comuns em microcontroladores de 32 bits, tais como os encontrados em PDAs (*Personal Digital Assistants*), celulares, tocadores de MP3 e na grande maioria de dispositivos eletrônicos do século XXI. A tecnologia desses microcontroladores tem melhorado com a incorporação de novas características de processamento e memória. Muitos deles já suportam diretamente o uso de sistemas operacionais, tais como o Android e o Linux.

A tecnologia Arduino, com sua base nos microcontroladores AVR, tem assumido grande destaque no meio acadêmico e industrial. O hardware, com a característica de conexão de outros módulos e a simplicidade de

programação, faz do Arduino um sucesso, permitindo que pessoas sem conhecimentos técnicos consigam programar e desenvolver os mais variados sistemas. Esse sucesso não é restrito somente a área educacional, muitos projetistas têm utilizado as características de hardware do Arduino para o desenvolvimento de projetos comerciais.

Para aproveitar o sucesso dos módulos (*shields*) feitos para o Arduino, têm surgido plataformas para a substituição do Arduino. Essas plataformas utilizam o mesmo formato dos conectores de expansão do Arduino, entretanto empregam outros microcontroladores, muitos já de 32 bits, com alta capacidade de processamento. Dessa maneira, como tudo é modular, o que deve mudar é a interface de programação e o conhecimento técnico; os módulos de expansão são os mesmos.

Os projetos propostos neste livro ilustram algumas possibilidades do emprego da tecnologia microcontrolada. O universo de técnicas para projetos eletrônicos com microcontroladores é muito vasto. Cada projeto exigirá soluções particulares e a crescente evolução tecnológica exige que o projetista se atualize constantemente.

25. REFERÊNCIAS

- BARBI, Ivo. **Projeto de Fontes Chaveadas.** 1a ed. Florianópolis: Ed. do autor, 2001.
- BOYLESTAD, Robert L.; NASHELSKY, Louis. **Dispositivos Eletrônicos e Teoria dos Circuitos.** 8 ed. São Paulo: Prentice Hall, 2004.
- BRAY, Jennifer; STURMAN, Charles F. **Bluetooth - Connect Without Cables.** 1 ed. New Jersey: Prentice Hall PTR, 2001.
- CARTER, Nicholas, **Computer Architecture.** 1 ed. New York: McGraw Hill, 2002.
- FITZGERALD, A. E.; KINGSLEY Jr., Charles; UMANS, Stephen D. **Máquinas Elétricas.** 6 ed. Porto Alegre: Bookman, 2006.
- GANSSLE, Jack et al. **Embedded Systems.** 1 ed. New York: Elsevier, 2008.
- HALLINAN, Christopher. **Embedded Linux Prime.** 2 ed. Boston: Prentice Hall, 2011.
- IBRAHIM, Dogan. **SD Card Projects Using the Pic Microcontroller.** 1 ed. New York, Elsevier, 2010.
- KOSOW, Irving I. **Máquinas Elétricas e Transformadores.** 15 ed. Porto Alegre: Globo, 1996.
- MARTINS, Denizar Cruz; BARBI, Ivo. **Introdução ao Estudo dos Conversores CC-CA.** 1 ed. Florianópolis: Ed. dos autores, 2005.
- NILSSON, James W.; RIEDEL, Susan A. **Circuitos Elétricos.** 6 ed. Rio de Janeiro: LTC, 2003.
- OLIVEIRA, R.; CARISSIMI, A.; TOSCANI, S. **Sistemas Operacionais.** 1 ed. Porto Alegre: Sagra-Luzzato, 2001.
- OPPENHEIM, Alan V.; SCHAFER, Ronald W.; BUCK, John R. **Discrete Signal Processing.** 2 ed. New Jersey: Prentice Hall, 1999.
- ORDONEZ, Edward D. M.; PENTEADO, Cesar G.; SILVA, Alexandre C. R. **Microcontroladores e FPGAs - Aplicações em Automação.** 1 ed. São Paulo: Novatec, 2006.
- PEREIRA, Fábio. **Microcontroladores PIC – Programação em C.** 7 ed. São Paulo: Érica, 2003.
- _____. **Microcontroladores MSP430 - Teoria e Prática.** 1 ed. São Paulo: Érica, 2005.
- _____. **Tecnologia ARM - Microcontroladores de 32 Bits.** 1 ed. São Paulo: Érica, 2007.

- PERTENCE Jr., Antônio. **Eletrônica Analógica - Amplificadores Operacionais e Filtros Ativos**. 7 ed. Porto Alegre: Artmed, 2012.
- SCHERZ, Paul. **Practical Electronics for Inventors**. 2 ed. New York, McGraw Hill, 2007.
- SCHILD'T, Herbert. **C Completo e Total**. 3 ed. São Paulo: Pearson Makron Books, 1997.
- SLOSS, Andrew; SYMES, Dominic; WRIGHT, Chris. **ARM System Developer's Guide: Designing and Optimizing System Software**. 1 ed. San Francisco: Morgan Kaufmann, 2004.
- SOUZA, Daniel Rodrigues. **Microcontroladores ARM7, o Poder dos 32 bits, Teoria e Prática**. 1 ed. São Paulo: Érica, 2006.
- SOUZA, David José; LAVÍNIA, Nicolas César. **Conectando o PIC - Recursos Avançados**. 4 ed. São Paulo: Érica, 2003.
- _____. **Desbravando o PIC**. 9 ed. São Paulo: Érica, 2005.
- STRUM, Robert D.; KIRK, Donald E. **First Principles of Discrete Systems and Digital Signal Processing**. 1 ed. New York: Addison Wesley, 1989.
- TANENBAUM, Andrew. S. **Sistemas Operacionais Modernos**. 1 ed. São Paulo: Prentice-Hall, 2004.
- _____.; WOODHULL, Albert S. **Sistemas Operacionais: Projeto e Implementação**. Trad. Edson Furmarkiewicz. 2. ed. Porto Alegre: Bookman, 2000.
- TOCCI, Ronald J.; WIDMER, Neal S.; MOSS, Gregory L. **Sistemas Digitais - Princípios e Aplicações**. 11 ed. São Paulo: Prentice Hall do Brasil, 2011.
- TORO, Vicente Del. **Fundamentos de Máquinas Elétricas**. 1 ed. Rio de Janeiro: LTC, 1999.
- WILLIAMS, Al. **Build Your Own Printed Circuit Board**. 1 ed. New York: McGraw Hill, 2004.
- YIU, Joseph. **The Definitive Guide to the ARM Cortex-M3**. 1 ed. New York: Elsevier, 2007.

Apostilas/tutoriais

AVR Assembler User Guide.

Introdução a Linguagem C. Fernanda Isabel Marques. Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina, 3 ed., Campus Florianópolis, Departamento Acadêmico de Eletrônica, 2009.

Introdução aos Microcontroladores. Marcos V. M. Villaça. Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina, 2 ed., Campus Florianópolis, Departamento Acadêmico de Eletrônica, 2007.

Real-Time Multitasking Executive for the 8051 Microcontroller User's Guide. KEIL software. 2002.

Manual de Referência do BRTOS – versão 1.50

PCB Design Tutorial. David Jones L. Revision A, June, 2009.

SD Specifications Part 1 - Physical Layer Simplified Specification Version 2.00, 25/09/2006. SD Group Technical Committee SD Card Association.

SD Card Reader Using the M9S08JM60 Series Designer Reference Manual. DRM104, Rev. 0, 07/2008.

The Insider's Guide to the NXP LPC2300/2400 Based Microcontrollers: An Engineer's Introduction to the LPC2300 & LPC2400 Series. Hitex Development Tools.

What's a Microcontroller? Student Guide version 3.0. Parallax inc.

Manuais de fabricantes

ATmega48/88/168/328/A/PA/P: Microcontroladores AVR

ATmega8(L): Microcontrolador AVR.

AT90S8515: Microcontrolador AVR

ITM-12864KOYTBL: LCD 128 × 64 pixels.

DS1307 - 64 x 8 Serial Real-Time Clock

DS18S20 - High-Precision 1-Wire Digital Thermometer

MAX6675 - Cold-Junction-Compensated K-Thermocouple-to-Digital Converter (0°C to +1024 °C)

LM35 - Precision Centigrade Temperature Sensors

TIL111 – Phototransistor Optoisolator

MOC3041 - 6-pin dip zero-cross optoisolators triac driver output

ULN2003 - Seven Darlington Arrays

TC72 – Digital Temperature Sensor with SPI Interface.

BT shield – Bluetooth to Serial Port Module Shield.

HC-SR04 – Ultrasonic Ranging Module.

Sítios

- www.atmel.com

Application Notes:

- AVR035 - Efficient C Coding for AVR
- AVR042 - AVR Hardware Design Considerations
- AVR121 – Enhancing ADC Resolution by Oversampling
- AVR122 – Calibration of the AVR’s Internal Temperature Reference
- AVR135 - Using Timer Capture to Measure PWM Duty Cycle
- AVR221 - Discrete PID Controller
- AVR243 - Matrix Keyboard Decoder
- AVR309 - Software Universal Serial Bus (USB)
- AVR311 - Using the TWI module as I2C slave
- AVR315 - Using the TWI module as I2C master
- AVR318 - Dallas 1-Wire® master
- AVR910 - In-System Programming

- www.microchip.com

Application Notes:

- Tips ‘n Tricks 8 pin PIC Microcontrollers
- AN512 - Implementing Ohmmeter/Temperature Sensor
- AN611 - Resistance and Capacitance Meter Using a PIC 16C622
- AN655 - D/A Conversion Using PWM and R-2R Ladders to Generate Sine and DTMF Waveforms
- AN1045 – Implementing File I/O Functions Using Microchip’s Memory Disk Drive File System Library

- www.nxp.com

Application Notes:

- AN10406 – Accessing SD/MMC card using SPI on LPC2000
- AN10369 - UART/SPI/I2C code examples

- www.arduino.cc
- www.avrfreaks.net
- www.retired.beyondlogic.org/pic/ringtones.htm
- www.bitmap2lcd.com
- www.ekitszone.com

- www.engbedded.com/fusecalc
- www.fairchildsemi.com
- www.futurlec.com
- www.hitex.co.uk
- www.hw-group.com
- www.intech-lcd.com
- www.keil.com
- www.labcenter.co.uk
- www.lancos.com/prog.html
- www.maxim-ic.com
- www.national.com
- www.parallax.com
- www.seekway.com.cn
- www.societyofrobots.com
- [www.techtoys.com.hk \(chp8_JHD12864LCD.pdf\)](http://www.techtoys.com.hk (chp8_JHD12864LCD.pdf))
- www.ti.com
- www.wikipedia.org
- alternatezone.com/electronics/files
- avr8-burn-o-mat.aaabbb.de
- code.google.com/p/brtos
- dkc1.digikey.com/us/EN/tod/CUI/BuzzerOverview/BuzzerOverview.html
- elm-chan.org/docs/mmc/mmc_e.html
- fritzing.org
- iteadstudio.com
- java.sun.com/javase/downloads/index.jsp

APÊNDICE

A. ASSEMBLY DO ATMEGA

A.1 INSTRUÇÕES DO ATMEGA

Mnemônico	Operandos	Descrição	Operação	Flags	Clocks
INSTRUÇÕES ARITMÉTICAS E LÓGICAS					
ADD	Rd, Rr	Soma dois registradores	$Rd \leftarrow Rd + Rr$	Z, C, N, V, H	1
ADC	Rd, Rr	Soma dois registradores com Carry	$Rd \leftarrow Rd + Rr + C$	Z, C, N, V, H	1
ADIW	Rdl, K	Soma o valor imediato à palavra (16 bits)	$Rdh:Rdl \leftarrow Rdh:Rdl + K$	Z, C, N, V, S	2
SUB	Rd, Rr	Subtrai dois registradores	$Rd \leftarrow Rd - Rr$	Z, C, N, V, H	1
SUBI	Rd, K	Subtrai uma constante do registrador	$Rd \leftarrow Rd - K$	Z, C, N, V, H	1
SBC	Rd, Rr	Subtrai dois registradores com Carry	$Rd \leftarrow Rd - Rr - C$	Z, C, N, V, H	1
SBCI	Rd, K	Subtrai constante de registrador com Carry	$Rd \leftarrow Rd - K - C$	Z, C, N, V, H	1
SBIW	Rdl, K	Subtrai valor imediato da palavra (16 bits)	$Rdh:Rdl \leftarrow Rdh:Rdl - K$	Z, C, N, V, S	2
AND	Rd, Rr	Lógica E entre registradores	$Rd \leftarrow Rd \bullet Rr$	Z, N, V	1
ANDI	Rd, K	Lógica E entre registrador e constante	$Rd \leftarrow Rd \bullet K$	Z, N, V	1
OR	Rd, Rr	Lógica OU entre registradores	$Rd \leftarrow Rd \vee Rr$	Z, N, V	1
ORI	Rd, K	Lógica OU entre registrador e constante	$Rd \leftarrow Rd \vee K$	Z, N, V	1
EOR	Rd, Rr	OU EXCLUSIVO entre registradores	$Rd \leftarrow Rd \otimes Rr$	Z, N, V	1
COM	Rd	Complemento de um	$Rd \leftarrow 0xFF - Rd$	Z, C, N, V	1
NEG	Rd	Complemento de dois	$Rd \leftarrow 0x00 - Rd$	Z, C, N, V, H	1
SBR	Rd, K	Ativa bit(s) no registrador	$Rd \leftarrow Rd \vee K$	Z, N, V	1
CBR	Rd, K	Limpa bit(s) no registrador	$Rd \leftarrow Rd \bullet (0xFF - K)$	Z, N, V	1
INC	Rd	Incrementa registrador	$Rd \leftarrow Rd + 1$	Z, N, V	1
DEC	Rd	Decrementa registrador	$Rd \leftarrow Rd - 1$	Z, N, V	1
TST	Rd	Teste de zero ou negativo	$Rd \leftarrow Rd \bullet Rr$	Z, N, V	1
CLR	Rd	Limpa registrador	$Rd \leftarrow Rd \otimes Rd$	Z, N, V	1
SER	Rd	Ativa registrador (todos os bits em 1)	$Rd \leftarrow 0xFF$	Nenhum	1
MUL	Rd, Rr	Multiplica sem sinal	$R1:R0 \leftarrow Rd \times Rr$	Z, C	2
MULS	Rd, Rr	Multiplica com sinal	$R1:R0 \leftarrow Rd \times Rr$	Z, C	2
MULSU	Rd, Rr	Multiplica registrador com sinal e sem sinal	$R1:R0 \leftarrow Rd \times Rr$	Z, C	2
FMUL	Rd, Rr	Multiplicação fracionária sem sinal	$R1:R0 \leftarrow (Rd \times Rr) << 1$	Z, C	2
FMULS	Rd, Rr	Multiplicação fracionária com sinal	$R1:R0 \leftarrow (Rd \times Rr) << 1$	Z, C	2
FMULSU	Rd, Rr	Multiplicação fracionária de registrador com sinal e sem sinal	$R1:R0 \leftarrow (Rd \times Rr) << 1$	Z, C	2

INSTRUÇÕES DE DESVIO					
RJMP	k	Desvio relativo	PC \leftarrow PC + k + 1	Nenhum	2
IJMP		Desvio indireto para (Z)	PC \leftarrow Z	Nenhum	2
RCALL	k	Chama de sub-rotina	PC \leftarrow PC + k + 1	Nenhum	3
ICALL		Chamada indireta para (Z)	PC \leftarrow Z	Nenhum	3
RET		Retorno de sub-rotina	PC \leftarrow STACK	Nenhum	4
RETI		Retorno de interrupção	PC \leftarrow STACK	I	4
CPSE	Rd, Rr	Compara, pula se igual	if(Rd=Rr) PC \leftarrow PC + 2 ou 3	Nenhum	1/ 2/ 3
CP	Rd, Rr	Compara	Rd - Rr	Z,N,V,C,H	1
CPC	Rd, Rr	Compara com Carry	Rd - Rr - C	Z,N,V,C,H	1
CPI	Rd, K	Compara registrador com valor imediato	Rd - K	Z,N,V,C,H	1
SBRC	Rr, b	Pula se o bit do registrador estiver limpo (0)	if(Rr(b)=0) PC \leftarrow PC + 2 ou 3	Nenhum	1/ 2/ 3
SBRS	Rr, b	Pula se o bit do registrador estiver ativo (1)	if(Rr(b)=1) PC \leftarrow PC + 2 ou 3	Nenhum	1/ 2/ 3
SBIC	P, b	Pula se o bit do registrador de I/O estiver limpo (0)	if(P(b)=0) PC \leftarrow PC + 2 ou 3	Nenhum	1/ 2/ 3
SBIS	P, b	Pula se o bit do registrador de I/O estiver ativo (1)	if(P(b)=1) PC \leftarrow PC + 2 ou 3	Nenhum	1/ 2/ 3
BRBS	s, k	Desvia se o bit de sinalização de status estiver ativo (1)	if(SREG(s)=1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRBC	s, k	Desvia se o bit de sinalização de status estiver limpo (0)	if(SREG(s)=0) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BREQ	k	Desvia se igual	if(Z=1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRNE	k	Desvia se diferente	if(Z=0) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRCS	k	Desvia se o bit de Carry estiver ativo (1)	if(C=1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRCC	k	Desvia se o bit de Carry estiver limpo (0)	if(C=0) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRSH	k	Desvia se igual ou maior	if(C=0) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRLO	k	Desvia se menor	if(C=1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRMI	k	Desvia se negativo	if(N=1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRPL	k	Desvia se positivo	if(N=0) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRGE	k	Desvia se maior ou igual, com sinal	if(N \otimes V = 0) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRLT	k	Desvia se menor que zero, com sinal	if(N \otimes V = 1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRHS	k	Desvia se o bit sinalizador de Carry auxiliar estiver ativo (1)	if(H=1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRHC	k	Desvia se o bit sinalizador de Carry auxiliar estiver limpo (0)	if(H=0) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRTS	k	Desvia se o bit sinalizador T estiver ativo (1)	if(T=1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRTC	k	Desvia se o bit sinalizador T estiver limpo (0)	if(T=0) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRVS	k	Desvia se o bit sinalizador de estouro estiver ativo (1)	if(V=1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRVC	k	Desvia se o bit sinalizador de estouro estiver limpo (0)	if(V=0) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRIE	k	Desvia se a interrupção estiver habilitada	if(I=1) PC \leftarrow PC + k + 1	Nenhum	1/ 2
BRID	k	Desvia se a interrupção estiver desabilitada	if(I=0) PC \leftarrow PC + k + 1	Nenhum	1/ 2

INSTRUÇÕES PARA A TRANSFERÊNCIA DE DADOS					
MOV	Rd, Rr	Movimento entre registradores	$Rd \leftarrow Rr$	Nenhum	1
MOVW	Rd, Rr	Copia registrador de palavra	$Rd + 1:Rd \leftarrow Rr + 1:Rr$	Nenhum	1
LDI	Rd, K	Carrega valor imediato	$Rd \leftarrow K$	Nenhum	1
LD	Rd, X	Carrega indiretamente	$Rd \leftarrow (X)$	Nenhum	2
LD	Rd, X+	Carrega indiretamente com pós-incremento	$Rd \leftarrow (X), X \leftarrow X + 1$	Nenhum	2
LD	Rd, -X	Carrega indiretamente com pré-decremento	$X \leftarrow X - 1, Rd \leftarrow (X)$	Nenhum	2
LD	Rd, Y	Carrega indiretamente	$Rd \leftarrow (Y)$	Nenhum	2
LD	Rd, Y+	Carrega indiretamente com pós-incremento	$Rd \leftarrow (Y), Y \leftarrow Y + 1$	Nenhum	2
LD	Rd, -Y	Carrega indiretamente com pré-decremento	$Y \leftarrow Y - 1, Rd \leftarrow (Y)$	Nenhum	2
LDD	Rd, Y+q	Carrega indiretamente com deslocamento	$Rd \leftarrow (Y + q)$	Nenhum	2
LD	Rd, Z	Carrega indiretamente	$Rd \leftarrow (Z)$	Nenhum	2
LD	Rd, Z+	Carrega indiretamente com pós-incremento	$Rd \leftarrow (Z), Z \leftarrow Z + 1$	Nenhum	2
LD	Rd, -Z	Carrega indiretamente com pré-decremento	$Z \leftarrow Z - 1, Rd \leftarrow (Z)$	Nenhum	2
LDD	Rd, Z+q	Carrega indiretamente com deslocamento	$Rd \leftarrow (Z + q)$	Nenhum	2
LDS	Rd, k	Carrega diretamente da SRAM	$Rd \leftarrow (k)$	Nenhum	2
ST	X, Rr	Armazena indiretamente	$(X) \leftarrow Rr$	Nenhum	2
ST	X+, Rr	Armazena indiretamente com pós-incremento	$(X) \leftarrow Rr, X \leftarrow X + 1$	Nenhum	2
ST	-X, Rr	Armazena indiretamente com pré-incremento	$X \leftarrow X - 1, (X) \leftarrow Rr$	Nenhum	2
ST	Y, Rr	Armazena indiretamente	$(Y) \leftarrow Rr$	Nenhum	2
ST	Y+, Rr	Armazena indiretamente com pós-incremento	$(Y) \leftarrow Rr, Y \leftarrow Y + 1$	Nenhum	2
ST	-Y, Rr	Armazena indiretamente com pré-incremento	$Y \leftarrow Y - 1, (Y) \leftarrow Rr$	Nenhum	2
STD	Y+q, Rr	Carrega indiretamente com deslocamento	$(Y + q) \leftarrow Rd$	Nenhum	2
ST	Z, Rr	Armazena indiretamente	$(Z) \leftarrow Rr$	Nenhum	2
ST	Z+, Rr	Armazena indiretamente com pós-incremento	$(Z) \leftarrow Rr, Z \leftarrow Z + 1$	Nenhum	2
ST	-Z, Rr	Armazena indiretamente com pré-incremento	$Z \leftarrow Z - 1, (Z) \leftarrow Rr$	Nenhum	2
STD	Z+q, Rr	Carrega indiretamente com deslocamento	$(Z + q) \leftarrow Rd$	Nenhum	2
STS	k, Rr	Carrega diretamente para SRAM	$(k) \leftarrow Rd$	Nenhum	2
LPM		Carrega a memória de programa	$R0 \leftarrow (Z)$	Nenhum	3
LPM	Rd, Z	Carrega a memória de programa	$Rd \leftarrow (Z)$	Nenhum	3
LPM	Rd, Z+	Carrega a memória de programa com pós-incremento	$Rd \leftarrow (Z), Z \leftarrow Z + 1$	Nenhum	3
SPM		Armazena a memória de programa	$(Z) \leftarrow R1:R0$	Nenhum	-
IN	Rd, P	Leitura de registrador de I/O	$Rd \leftarrow P$	Nenhum	1
OUT	P, Rr	Escrita de registrador de I/O	$P \leftarrow Rr$	Nenhum	1
PUSH	Rr	Coloca registrador na pilha (<i>Stack</i>)	$STACK \leftarrow Rr$	Nenhum	2
POP	Rd	Retira registrador da pilha (<i>Stack</i>)	$Rd \leftarrow STACK$	Nenhum	2

INSTRUÇÕES DE BIT E TESTE DE BIT						
SBI	P, b	Ativa o bit no registrador de I/O	I/O(P,b) \leftarrow 1	Nenhum	2	
CBI	P, b	Limpa o bit do registrador de I/O	I/O(P,b) \leftarrow 0	Nenhum	2	
LSL	Rd	Deslocamento lógico à esquerda	Rd(n+1) \leftarrow Rd(n), Rd(0) \leftarrow 0	Z, C, N, V	1	
LSR	Rd	Deslocamento lógico à direita	Rd(n) \leftarrow Rd(n+1), Rd(7) \leftarrow 0	Z, C, N, V	1	
ROL	Rd	Rotação à esquerda através do <i>Carry</i>	Rd(0) \leftarrow C, Rd(n+1) \leftarrow Rd(n), C \leftarrow Rd(7)	Z, C, N, V	1	
ROR	Rd	Rotação à direita através do <i>Carry</i>	Rd(7) \leftarrow C, Rd(n) \leftarrow Rd(n+1), C \leftarrow Rd(0)	Z, C, N, V	1	
ASR	Rd	Deslocamento aritmético à direita	Rd(n) \leftarrow Rd(n+1), n=0..6	Z, C, N, V	1	
SWAP	Rd	Troca nibbles (4 bits)	Rd(3..0) \leftarrow Rd(7..4), Rd(7..4) \leftarrow Rd(3..0)	Nenhum	1	
BSET	s	Ativa bit (<i>flag</i>)	SREG(s) \leftarrow 1	SREG(s)	1	
BCLR	s	Limpa bit (<i>flag</i>)	SREG(s) \leftarrow 0	SREG(s)	1	
BST	Rr, b	Armazenagem do bit do registrador para o T	T \leftarrow Rr(b)	T	1	
BLD	Rd, b	Carga do bit T para o registrador	Rd(b) \leftarrow T	Nenhum	1	
SEC		Ativa o <i>Carry</i>	C \leftarrow 1	C	1	
CLC		Limpa o <i>Carry</i>	C \leftarrow 0	C	1	
SEN		Ativa o bit de negativo	N \leftarrow 1	N	1	
CLN		Limpa o bit de negativo	N \leftarrow 0	N	1	
SEZ		Ativa o bit de zero	Z \leftarrow 1	Z	1	
CLZ		Limpa o bit de zero	Z \leftarrow 0	Z	1	
SEI		Habilita a interrupção global	I \leftarrow 1	I	1	
CLI		Desabilita interrupção global	I \leftarrow 0	I	1	
SES		Ativa o bit de teste de sinal	S \leftarrow 1	S	1	
CLS		Desativa o bit de teste de sinal	S \leftarrow 0	S	1	
SEV		Ativa o bit de estouro do complemento de dois	V \leftarrow 1	V	1	
CLV		Limpa o bit de estouro do complemento de dois	V \leftarrow 0	V	1	
SET		Ativa o bit T	T \leftarrow 1	T	1	
CLT		Limpa o bit T	T \leftarrow 0	T	1	
SEH		Ativa o bit de <i>Carry auxiliar</i>	H \leftarrow 1	H	1	
CLH		Limpa o bit de <i>Carry auxiliar</i>	H \leftarrow 0	H	1	

INSTRUÇÕES DE CONTROLE DA UNIDADE MICROCONTROLADA

NOP	Nenhuma operação		Nenhum	1
SLEEP	Entra no modo <i>sleep</i>	Consultar manual do fabricante	Nenhum	1
WDR	Reset do <i>Watchdog</i>	Consultar manual do fabricante	Nenhum	1

- Ativação de bit significa colocá-lo em 1.
- Limpeza de bit significa colocá-lo em 0.

Legenda:

- Rd = registrador de destino (e origem).
- Rr = registrador de origem.
- s, b = constante (0..7); pode ser uma expressão constante.
- K = constante, faixa de valores depende da instrução; pode ser uma expressão constante.
- P = constante (0..31/63), pode ser uma expressão constante ou um registrador de I/O.
- q = constante (0..63), pode ser uma expressão constante.
- Rdl = R24, R26, R28, R30, usado nas instruções ADIW e SBIW.
- X, Y, Z = registradores de 16 bits para endereçamento indireto (X=R27:26, Y=R29:28 e Z=R31:30).

Todos as instruções que operam com registradores de uso geral tem acesso direto em um único ciclo a todos eles. As exceções são as cinco instruções lógicas e aritméticas entre uma constante e um registrador, SBCI, SUBI, CPI, ANDI e ORI, e a instrução para carga de constante imediata, LDI. Essas instruções se aplicam somente a metade superior dos registradores de uso geral (R16..R31). As instruções SBC, SUB, CP, AND, OR e as demais operações entre um ou dois registradores se aplicam a todo o banco de registradores.

A.2 STATUS REGISTER

Invisível para programadores C, seu conhecimento é imprescindível para programadores *Assembly*.

Bit	7	6	5	4	3	2	1	0
SREG	I	T	H	S	V	N	Z	C
Lê/Escrive	L/E							
Valor Inicial	0	0	0	0	0	0	0	0

Bit 7 – I – Global Interrupt Enable

Este bit é a chave geral para habilitar as interrupções. Cada interrupção individual possui seus registradores de controle. O bit **I** é limpo quando uma interrupção ocorre (impedindo que outras ocorram simultaneamente) e volta a ser ativo quando se termina o tratamento da interrupção (instrução RETI).

Bit 6 – T – Bit Copy Storage

Serve para copiar o valor de um bit de um registrador ou escrever o valor de um bit em um registrador (instruções BLD e BST).

Bit 5 – H – Half Carry Flag

Indica quando um *Carry* auxiliar (em um *nibble*) ocorreu em alguma operação aritmética. Útil em aritmética BCD.

Bit 4 – S – Sign Bit, $S = N \oplus V$

O bit S é o resultado de um ou exclusivo entre o *flag* negativo **N** e o *flag* de estouro do complemento de dois **V**.

Bit 3 – V – Two's Complement Overflow Flag

O *flag* de estouro do complemento de dois ajuda na aritmética em complemento de dois.

Bit2 – N – Negative Flag

O *flag* negativo indica quando uma operação aritmética ou lógica resulta em um valor negativo.

Bit1 – Z – Zero Flag

O *flag* zero indica quando uma operação aritmética ou lógica resulta em zero.

Bit 0 – C – Carry Flag

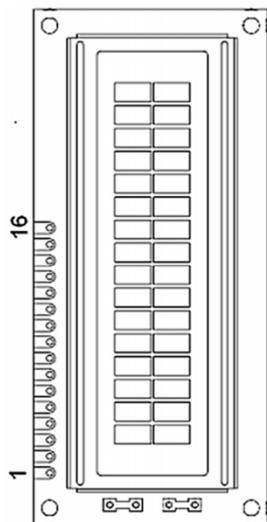
O *flag* de *Carry* indica quando houve *Carry* (vai 1) numa operação aritmética.

B. DISPLAY DE CRISTAL LÍQUIDO 16 x 2 - CONTROLADOR HD44780

B.1 PINAGEM

A pinagem do LCD 16×2 geralmente segue o padrão abaixo. Entretanto, alguns fabricantes podem inverter a ordem dos pinos (recomenda-se a consulta ao manual do fabricante).

Tab. B1: Pinagem de um LCD 16×2.



Pino	Função	Descrição
1	Alimentação	VSS (GND)
2	Alimentação	VCC
3	VEE	Tensão para ajuste do contraste do LCD
4	RS	<i>Register Select</i> : 1 = dado, 0 = instrução
5	R/W	<i>Read/Write</i> : 1 = leitura, 0 = escrita
6	E	<i>Enable</i> : 1 = habilita, 0 = desabilita
7	DB0	Barramento de dados
8	DB1	
9	DB2	
10	DB3	
11	DB4	
12	DB5	
13	DB6	
14	DB7	
15	LED+ (A)	Anodo do LED de iluminação de fundo
16	LED - (K)	Catodo do LED de iluminação de fundo

B.2 CÓDIGOS DE INSTRUÇÕES

Tab. B2: Detalhamento do códigos de instruções.

INSTRUÇÃO	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		Descrição	Execução
Limpa Display	0	0	0	0	0	0	0	0	0	1		Limpa todo o display e retorna o cursor para a primeira posição da primeira linha.	1,6 ms
Retorno do cursor	0	0	0	0	0	0	0	0	1	-		Retorna o cursor para a 1ª coluna da 1ª linha. Retorna a mensagem previamente deslocada a sua posição original.	1,6 ms
Fixa o modo de Funcionamento	0	0	0	0	0	0	0	1	X	S		Ajusta o sentido do deslocamento do cursor (X=0 p/ a esquerda, X=1 p/ a direita). Determina se a mensagem deve ou não ser deslocada com a entrada de um novo caractere (S = 1, SIM). Esta instrução tem efeito somente durante a leitura e escrita de dados.	40 µs
Controle do Display	0	0	0	0	0	0	1	D	C	B		Liga (D=1) ou desliga display (D=0). Liga (C=1) ou desliga cursor (C=0). Cursor piscante (B=1) se C=1.	40 µs
Desloca cursor ou mensagem	0	0	0	0	0	1	C	R	-	-		Desloca o cursor (C=0) ou a mensagem (C=1) para a direita se R=1 ou esquerda se R=0. Desloca sem alterar o conteúdo da DDRAM	40 µs
Fixa modo de utilização do módulo LCD	0	0	0	0	1	Y	N	F	-	-		Comunicação do módulo com 8 bits (Y=1) ou 4 bits (Y=0). Número de linhas: 1 (N=0) e 2 ou mais (N=1). Matriz do caractere: 5x7 (F=0) ou 5x10 (F=1). Esta instrução deve ser empregada na inicialização.	40 µs
Endereço da CGRAM	0	0	0	1								Fixa o endereço da CGRAM para posterior envio ou leitura de um dado (byte).	40 µs
Endereço da DDRAM	0	0	1									Fixa o endereço da DDRAM para posterior envio ou leitura de um dado (byte).	40 µs
Leitura do bit de ocupado e do conteúdo de endereços	0	1	B F									Lê o conteúdo do contador de endereços AC e o BF. O bit 7 do BF indica se a última operação foi concluída (BF=0 concluída, BF=1 em execução).	-
Escreve dado na CGRAM/DDRAM	1	0										Grava o byte presente nos pinos de dados no local apontado pelo contador de endereços (posição do cursor).	40 µs
Lê dado da CGRAM/DDRAM	1	1										Lê o byte do local apontado pelo contador de endereços (posição do cursor).	40 µs

Tab. B3: Resumo dos códigos de instruções.

Descrição	Modo	Código Hexa
Controle do display	Liga (sem cursor)	0x0C
	Desliga	0x0A/0x08
Limpa display com retorno do cursor		0x01
Controle do cursor	Liga	0x0E
	Desliga	0x0C
	Desloca p/ a esquerda	0x10
	Desloca p/ a direita	0x14
	Retorno	0x02
	Cursor piscante	0x0D
	Cursor com alternância	0x0F
Sentido de deslocamento do cursor na entrada de um caractere	Para a esquerda	0x04
	Para a direita	0x06
Deslocamento da mensagem na entrada de um caractere	Para a esquerda	0x07
	Para a direita	0x05
Deslocamento da mensagem sem a entrada de caractere	Para a esquerda	0x18
	Para a direita	0x1C
Endereço da primeira posição do cursor	Primeira linha	0x80
	Segunda linha	0xC0

B.3 ENDEREÇO DOS SEGMENTOS (DDRAM)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Linha 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
Linha 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF

B.4 CONJUNTO E CÓDIGO DOS CARACTERES

Os principais caracteres reconhecidos pelo módulo LCD seguem o código ASCII. Todavia, existem outros caracteres. Na tabela B.4, encontra-se o conjunto total de caracteres que podem ser apresentados. O código a ser enviado ao LCD é obtido concatenando-se o valor horizontal da parte superior da tabela com o valor vertical do lado esquerdo (o *nibble* alto com

o *nibble* baixo). Por exemplo, para o caractere **L**, o *nibble* alto é **4**_ e o *nibble* baixo é **_C**, o que resulta no valor hexadecimal 0x4C.

Na tabela B.4, pode-se, ainda, observar os endereços dos 8 caracteres que podem ser criados na CGRAM (0x00 até 0x07).

Tab. B4: Conjunto dos caracteres para um LCD 16 × 2.

NIBBLE ALTO \ NIBBLE BAIXO	0_	1_	2_	3_	4_	5_	6_	7_	8_	9_	A_	B_	C_	D_	E_	F_
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
_0	xxxx 0000 CG RAM (1)	0000 0000	0000 0001	0000 0010	0000 0011	0000 0100	0000 0101	0000 0110	0000 0111	0000 1000	0000 1001	0000 1010	0000 1011	0000 1100	0000 1101	0000 1110
_1	xxxx 0001 CG RAM (2)	0001 0000	0001 0001	0001 0010	0001 0011	0001 0100	0001 0101	0001 0110	0001 0111	0001 1000	0001 1001	0001 1010	0001 1011	0001 1100	0001 1101	0001 1110
_2	xxxx 0010 CG RAM (3)	0010 0000	0010 0001	0010 0010	0010 0011	0010 0100	0010 0101	0010 0110	0010 0111	0010 1000	0010 1001	0010 1010	0010 1011	0010 1100	0010 1101	0010 1110
_3	xxxx 0011 CG RAM (4)	0011 0000	0011 0001	0011 0010	0011 0011	0011 0100	0011 0101	0011 0110	0011 0111	0011 1000	0011 1001	0011 1010	0011 1011	0011 1100	0011 1101	0011 1110
_4	xxxx 0100 CG RAM (5)	0100 0000	0100 0001	0100 0010	0100 0011	0100 0100	0100 0101	0100 0110	0100 0111	0100 1000	0100 1001	0100 1010	0100 1011	0100 1100	0100 1101	0100 1110
_5	xxxx 0101 CG RAM (6)	0101 0000	0101 0001	0101 0010	0101 0011	0101 0100	0101 0101	0101 0110	0101 0111	0101 1000	0101 1001	0101 1010	0101 1011	0101 1100	0101 1101	0101 1110
_6	xxxx 0110 CG RAM (7)	0110 0000	0110 0001	0110 0010	0110 0011	0110 0100	0110 0101	0110 0110	0110 0111	0110 1000	0110 1001	0110 1010	0110 1011	0110 1100	0110 1101	0110 1110
_7	xxxx 0111 CG RAM (8)	0111 0000	0111 0001	0111 0010	0111 0011	0111 0100	0111 0101	0111 0110	0111 0111	0111 1000	0111 1001	0111 1010	0111 1011	0111 1100	0111 1101	0111 1110
_8	xxxx 1000 (1)	1000 0000	1000 0001	1000 0010	1000 0011	1000 0100	1000 0101	1000 0110	1000 0111	1000 1000	1000 1001	1000 1010	1000 1011	1000 1100	1000 1101	1000 1110
_9	xxxx 1001 (2)	1001 0000	1001 0001	1001 0010	1001 0011	1001 0100	1001 0101	1001 0110	1001 0111	1001 1000	1001 1001	1001 1010	1001 1011	1001 1100	1001 1101	1001 1110
_A	xxxx 1010 (3)	1010 0000	1010 0001	1010 0010	1010 0011	1010 0100	1010 0101	1010 0110	1010 0111	1010 1000	1010 1001	1010 1010	1010 1011	1010 1100	1010 1101	1010 1110
_B	xxxx 1011 (4)	1011 0000	1011 0001	1011 0010	1011 0011	1011 0100	1011 0101	1011 0110	1011 0111	1011 1000	1011 1001	1011 1010	1011 1011	1011 1100	1011 1101	1011 1110
_C	xxxx 1100 (5)	1100 0000	1100 0001	1100 0010	1100 0011	1100 0100	1100 0101	1100 0110	1100 0111	1100 1000	1100 1001	1100 1010	1100 1011	1100 1100	1100 1101	1100 1110
_D	xxxx 1101 (6)	1101 0000	1101 0001	1101 0010	1101 0011	1101 0100	1101 0101	1101 0110	1101 0111	1101 1000	1101 1001	1101 1010	1101 1011	1101 1100	1101 1101	1101 1110
_E	xxxx 1110 (7)	1110 0000	1110 0001	1110 0010	1110 0011	1110 0100	1110 0101	1110 0110	1110 0111	1110 1000	1110 1001	1110 1010	1110 1011	1110 1100	1110 1101	1110 1110
_F	xxxx 1111 (8)	1111 0000	1111 0001	1111 0010	1111 0011	1111 0100	1111 0101	1111 0110	1111 0111	1111 1000	1111 1001	1111 1010	1111 1011	1111 1100	1111 1101	1111 1110

C. ERROS E AJUSTE DA TAXA DE COMUNICAÇÃO DA USART

A faixa de operação para a taxa de comunicação da USART é dependente do descasamento entre a taxa dos bits recebidos e a taxa interna de geração dos bits. Se o transmissor envia *frames* muito rapidamente ou muito devagar, ou se o gerador interno da taxa de transmissão do ATmega não tem uma base de frequência similar, o receptor não será capaz de sincronizar os *frames* com o bit de início da comunicação.

Os erros associados às diferentes taxas de comunicação e os valores para ajuste do registrador UBRR0 para obter a taxa de comunicação desejada são apresentados na tab. C.1. Os valores são baseados em frequências comumente encontradas em osciladores.

Tab. C.1 – Exemplos de valores para o UBRR0 para frequências comuns de osciladores.

Taxa de comunicação (bps)	$f_{osc} = 1,0000 \text{ MHz}$				$f_{osc} = 1,8432 \text{ MHz}$			
	U2X0=0		U2X0=1		U2X0=0		U2X0=1	
	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro
2400	25	0,2%	51	0,2%	47	0,0%	95	0,0%
4800	12	0,2%	25	0,2%	23	0,0%	47	0,0%
9600	6	-7,0%	12	0,2%	11	0,0%	23	0,0%
14,4 k	3	8,5%	8	-3,5%	7	0,0%	15	0,0%
19,2 k	2	8,5%	6	-7,0%	5	0,0%	11	0,0%
28,8 k	1	8,5%	3	8,5%	3	0,0%	7	0,0%
38,4 k	1	-18,6%	2	8,5%	2	0,0%	5	0,0%
57,6 k	0	8,5%	1	8,5%	1	0,0%	3	0,0%
76,8 k	-	-	1	-18,6%	1	-25,0%	2	0,0%
115,2 k	-	-	0	8,5%	0	0,0%	1	0,0%
230,4 k	-	-	-	-	-	-	0	0,0%
250 k	-	-	-	-	-	-	-	-
Máximo	62,5 kbps		125 kbps		115,2 kbps		230,4 kbps	

Taxa de comunicação (bps)	$f_{osc} = 2,0000$ MHz				$f_{osc} = 3,6864$ MHz			
	U2X0=0		U2X0=1		U2X0=0		U2X0=1	
	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro
2400	51	0,2%	103	0,2%	95	0,0%	191	0,0%
4800	25	0,2%	51	0,2%	47	0,0%	95	0,0%
9600	12	0,2%	25	0,2%	23	0,0%	47	0,0%
14,4 k	8	-3,5%	16	2,1%	15	0,0%	31	0,0%
19,2 k	6	-7,0%	12	0,2%	11	0,0%	23	0,0%
28,8 k	3	8,5%	8	-3,5%	7	0,0%	15	0,0%
38,4 k	2	8,5%	6	-7,0%	5	0,0%	11	0,0%
57,6 k	1	8,5%	3	8,5%	3	0,0%	7	0,0%
76,8 k	1	-18,6%	2	8,5%	2	0,0%	5	0,0%
115,2 k	0	8,5%	1	8,5%	1	0,0%	3	0,0%
230,4 k	-	-	-	-	0	0,0%	1	0,0%
250 k	-	-	0	0,0%	0	-7,8%	1	-7,8%
0,5 M	-	-	-	-	-	-	0	-7,8%
Máximo	125 kbps		250 kbps		230,4 kbps		460,8 kbps	

Taxa de comunicação (bps)	$f_{osc} = 4,0000$ MHz				$f_{osc} = 7,3728$ MHz			
	U2X0=0		U2X0=1		U2X0=0		U2X0=1	
	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro
2400	103	0,2%	207	0,2%	191	0,0%	383	0,0%
4800	51	0,2%	103	0,2%	95	0,0%	191	0,0%
9600	25	0,2%	51	0,2%	47	0,0%	95	0,0%
14,4 k	16	2,1%	34	-0,8%	31	0,0%	63	0,0%
19,2 k	12	0,2%	25	0,2%	23	0,0%	47	0,0%
28,8 k	8	-3,5%	16	2,1%	15	0,0%	31	0,0%
38,4 k	6	-7,0%	12	0,2%	11	0,0%	23	0,0%
57,6 k	3	8,5%	8	-3,5%	7	0,0%	15	0,0%
76,8 k	2	8,5%	6	-7,0%	5	0,0%	11	0,0%
115,2 k	1	8,5%	3	8,5%	3	0,0%	7	0,0%
230,4 k	0	8,5%	1	8,5%	1	0,0%	3	0,0%
250 k	0	0,0%	1	0,0%	1	-7,8%	3	-7,8%
0,5 M	-	-	0	0,0%	0	-7,8%	1	-7,8%
1 M	-	-	-	-	-	-	0	-7,8%
Máximo	250 kbps		0,5 Mbps		460,8 kbps		921,6 kbps	

Taxa de comunicação (bps)	$f_{osc} = 8,0000$ MHz				$f_{osc} = 11,0592$ MHz			
	U2X0=0		U2X0=1		U2X0=0		U2X0=1	
	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro
2400	207	0,2%	416	-0,1%	287	0,0%	575	0,0%
4800	103	0,2%	207	0,2%	143	0,0%	287	0,0%
9600	51	0,2%	103	0,2%	71	0,0%	143	0,0%
14,4 k	34	-0,8%	68	0,6%	47	0,0%	95	0,0%
19,2 k	25	0,2%	51	0,2%	35	0,0%	71	0,0%
28,8 k	16	2,1%	34	-0,8%	23	0,0%	47	0,0%
38,4 k	12	0,2%	25	0,2%	17	0,0%	35	0,0%
57,6 k	8	-3,5%	16	2,1%	11	0,0%	23	0,0%
76,8 k	6	-7,0%	12	0,2%	8	0,0%	17	0,0%
115,2 k	3	8,5%	8	-3,5%	5	0,0%	11	0,0%
230,4 k	1	8,5%	3	8,5%	2	0,0%	5	0,0%
250 k	1	0,0%	3	0,0%	2	-7,8%	5	-7,8%
0,5 M	0	0,0%	1	0,0%	-	-	2	-7,8%
1 M	-	-	0	0,0%	-	-	-	-
Máximo	0,5 Mbps		1 Mbps		691,2 kbps		1,3824 Mbps	

Taxa de comunicação (bps)	$f_{osc} = 14,7456$ MHz				$f_{osc} = 16,0000$ MHz			
	U2X0=0		U2X0=1		U2X0=0		U2X0=1	
	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro
2400	383	0,0%	767	0,0%	416	-0,1%	832	0,0%
4800	191	0,0%	383	0,0%	207	0,2%	416	-0,1%
9600	95	0,0%	191	0,0%	103	0,2%	207	0,2%
14,4 k	63	0,0%	127	0,0%	68	0,6%	138	-0,1%
19,2 k	47	0,0%	95	0,0%	51	0,2%	103	0,2%
28,8 k	31	0,0%	63	0,0%	34	-0,8%	68	0,6%
38,4 k	23	0,0%	47	0,0%	25	0,2%	51	0,2%
57,6 k	15	0,0%	31	0,0%	16	2,1%	34	-0,8%
76,8 k	11	0,0%	23	0,0%	12	0,2%	25	0,2%
115,2 k	7	0,0%	15	0,0%	8	-3,5%	16	2,1%
230,4 k	3	0,0%	7	0,0%	3	8,5%	8	-3,5%
250 k	3	-7,8%	6	5,3%	3	0,0%	7	0,0%
0,5 M	1	-7,8%	3	-7,8%	1	0,0%	3	0,0%
1 M	0	-7,8%	1	-7,8%	0	0,0%	1	0,0%
Máximo	921,6 kbps		1,8432 Mbps		1 Mbps		2 Mbps	

Taxa de comunicação (bps)	$f_{osc} = 18,4320 \text{ MHz}$				$f_{osc} = 20,0000 \text{ MHz}$			
	U2X0=0		U2X0=1		U2X0=0		U2X0=1	
	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro	UBRR0	Erro
2400	479	0,0%	959	0,0%	520	0,0%	1041	0,0%
4800	239	0,0%	479	0,0%	259	0,2%	520	0,0%
9600	119	0,0%	239	0,0%	129	0,2%	259	0,2%
14,4 k	79	0,0%	159	0,0%	86	-0,2%	173	-0,2%
19,2 k	59	0,0%	119	0,0%	64	0,2%	129	0,2%
28,8 k	39	0,0%	79	0,0%	42	0,9%	86	-0,2%
38,4 k	29	0,0%	59	0,0%	32	-1,4%	64	0,2%
57,6 k	19	0,0%	39	0,0%	21	-1,4%	42	0,9%
76,8 k	14	0,0%	29	0,0%	15	1,7%	32	-1,4%
115,2 k	9	0,0%	19	0,0%	10	-1,4%	21	-1,4%
230,4 k	4	0,0%	9	0,0%	4	8,5%	10	-1,4%
250 k	4	-7,8%	8	2,4%	4	0,0%	9	0,0%
0,5 M	-	-	4	-7,8%	-	-	4	0,0%
1 M	-	-	-	-	-	-	-	-
Máximo	1,152 Mbps		2,304 Mbps		1,25 Mbps		2,5 Mbps	

D. CIRCUITOS PARA O ACIONAMENTO DE CARGAS

A corrente de saída que os circuitos digitais podem fornecer para dispositivos externos geralmente é insuficiente para a maioria das cargas, tais como: relés, motores e lâmpadas. Quando um microcontrolador é utilizado para o acionamento de alguma carga com corrente superior ao que seu pino pode suprir ou drenar, deve-se empregar um circuito específico para fazer a ligação entre o microcontrolador e a carga. Esses circuitos costumam ser designados em eletrônica como *buffers* de corrente ou *drivers* de corrente. Da mesma forma, pode ser necessário adequar ou isolar um sistema digital para o trabalho com níveis de tensões diferentes e, nesse caso, empregam-se circuitos isoladores ou acopladores. A seguir, serão apresentadas algumas possibilidades básicas de interfaces entre circuitos digitais e o mundo analógico externo.

D.1 A CHAVE TRANSISTORIZADA

A chave transistorizada é muito utilizada para acionamentos e inversão de sinais digitais. Seu projeto é fácil, bastando saber: a corrente de carga, o ganho do transistor e a tensão de acionamento. Na fig. D.1, são ilustradas duas chaves transistorizadas, uma com um transistor NPN e a outra com um PNP, ambos devidamente polarizados.

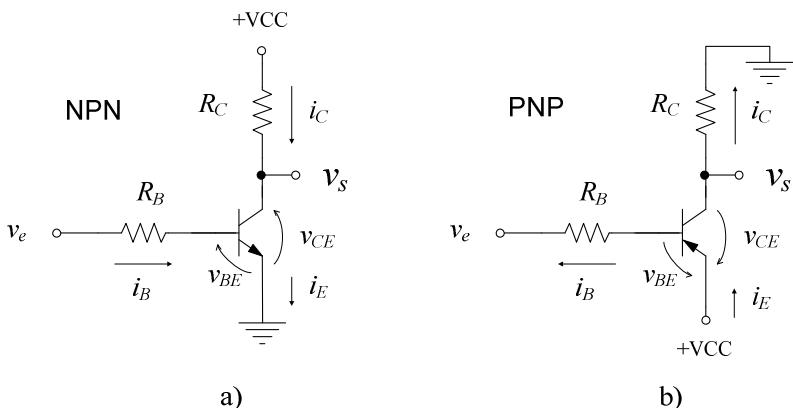


Fig. D.1 – Chaves transistorizadas: a) transistor NPN e b) transistor PNP.

O transistor é um dispositivo eletrônico muitíssimo utilizado⁵⁷. É comumente modelado como uma fonte de corrente controlada por corrente. A corrente de base controla a corrente de coletor, por um fator β (ou h_{FE}), chamado ganho de corrente ($i_C = \beta \cdot i_B$). O transistor quando operando como amplificador deve trabalhar na região ativa, onde variações na corrente de base acarretam variações na corrente de coletor e a tensão entre coletor e emissor pode variar (v_{CE}).

Quando operando como chave, ou seja, quando a corrente de carga é limitada apenas por sua própria resistência, o transistor trabalha saturado e a relação entre a corrente de coletor e de base deixa de ser linear (se a corrente de base for aumentada, a corrente de coletor não se altera). Na saturação, a corrente de base garante uma tensão mínima entre o coletor e o emissor (quase zero na prática) e a resistência do transistor se torna pequena.

⁵⁷ O estudo detalhado do transistor está além do escopo deste livro. Para maiores detalhes ver: BOYLESTAD, Robert L.; NASHELSKY, Louis. **Dispositivos Eletrônicos e Teoria dos Circuitos**. 8 ed. São Paulo: Prentice Hall, 2004.

Considerando uma fonte assimétrica (fig. D.1), o transistor trabalhará com tensão positiva em relação ao terra. Analisando a fig. D.1a para o dimensionamento de uma chave transistorizada, as variáveis R_C , v_e e o ganho do transistor NPN são dados fixos do circuito, restando calcular apenas R_B . Para esse cálculo, deve-se empregar: o ganho mínimo do transistor (β_{\min}), a tensão entre base e emissor (v_{BE}) de aproximadamente 0,7 V (tensão de um diodo de silício diretamente polarizado) e a tensão entre coletor e emissor (v_{CE}) de aproximadamente 0 V (saturação). Além disso, se calcula a corrente de base empregando-se um coeficiente de segurança de 2 vezes para garantir a saturação. Assim, tem-se:

1. Cálculo da corrente de carga:

$$i_C = \frac{V_{CC} - v_{CE}}{R_C} \rightarrow i_C = \frac{V_{CC}}{R_C}$$

2. Cálculo da corrente de base:

$$i_B = \frac{2i_C}{\beta_{\min}}$$

Onde se multiplica a corrente de coletor por dois. Esse é o fator de segurança que aliado ao ganho mínimo garante a saturação.

3. Cálculo de R_B :

$$R_B = \frac{v_e - v_{BE}}{i_B} \rightarrow R_B = \frac{v_e - 0,7}{i_B} [\Omega]$$

Quando houver necessidade do arredondamento dos números, sempre se deve favorecer o aumento da corrente de base.

Exemplo D.1: Deseja-se acionar o LED do circuito da fig. D.2 com um sinal de 5 V. O LED deve operar com uma corrente de 20 mA (queda de 2 V sobre o LED). Será empregado o transistor NPN de chaveamento 2N2222 que tem um ganho mínimo de 50. Determinar R_D , R_B e a tensão v_s de acordo com a tensão de entrada.

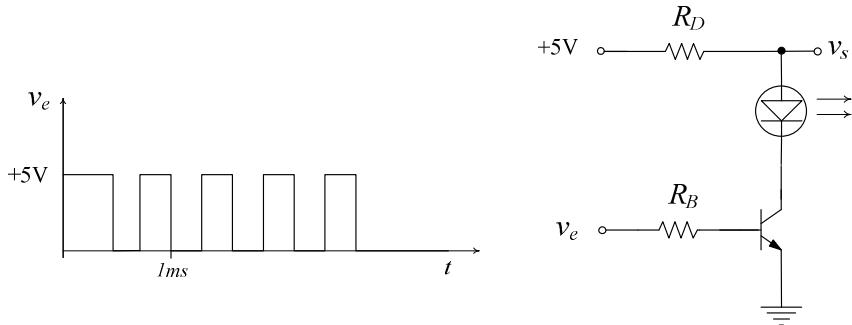


Fig. D.2 – Circuito para o cálculo da chave transistorizada do exemplo D.1.

A tensão em R_D , com $v_{CE} \approx 0$ V e $v_{LED} = 2$ V, é:

$$v_D = 5 - 2 = 3 \text{ V}$$

Como a corrente no LED é a mesma de R_D , calcula-se R_D :

$$R_D = \frac{v_D}{i_{LED}} \quad \rightarrow \quad R_D = \frac{3}{0,02} = 150 \Omega$$

Calculando i_B e R_B :

$$i_B = \frac{2 \times 0,02}{50} \rightarrow i_B = 0,8 \text{ mA} \rightarrow i_B = 1 \text{ mA} \text{ (com arredondamento)}$$

$$R_B = \frac{5 - 0,7}{0,001} = 4,3 \text{ k}\Omega \quad (\text{valor comercial de resistência})$$

A tensão v_s é ilustrada na fig. D.3.

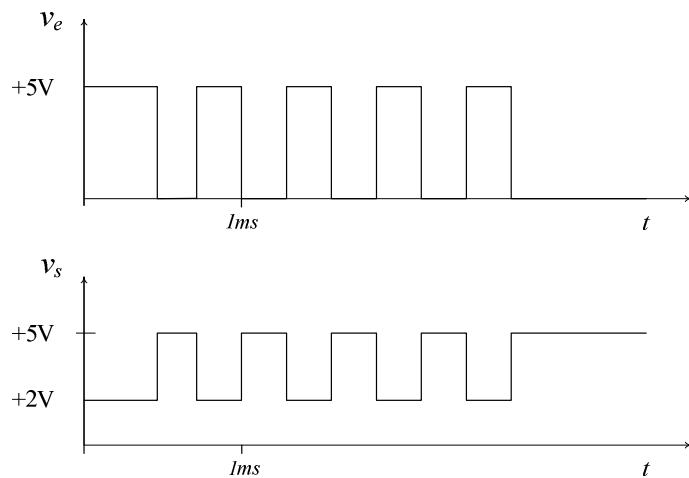


Fig. D.3 – Tensão aplicada no resistor de base e tensão resultante no anodo do LED para o circuito da fig. D2.

A tensão v_e aplicada no transistor liga o LED. O resistor R_D é necessário para limitar a corrente no LED e impedir que ele seja danificado.

Exemplo D.2: Deseja-se acionar uma carga que consome 100 mA quando ligada à chave transistorizada da fig. D.4. É empregado o transistor PNP de chaveamento 2N2907 que têm um ganho mínimo de 50. Determinar R_B e a tensão v_s de acordo com a tensão de entrada.

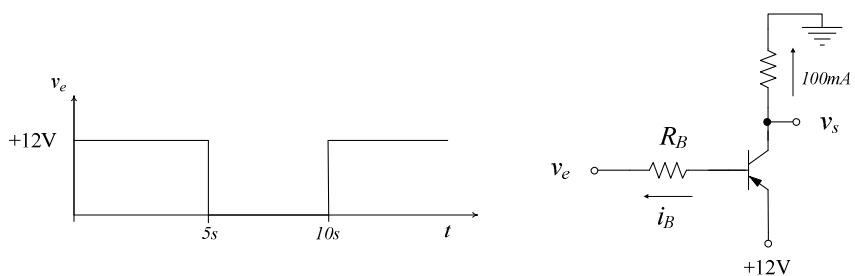


Fig. D.4 – Circuito para cálculo da chave transistorizada do exemplo D.2.

Apesar da inversão de polaridade na alimentação do 2N2907, a forma de cálculo da chave mantém-se igual, com a diferença que a chave é acionada com tensão zero.

Calculando i_B e R_B :

$$i_B = \frac{2 \times 0,1}{50} \rightarrow i_B = 4 \text{ mA}$$

$$R_B = \frac{12 - 0,7}{0,004} = 2825 \Omega \rightarrow R_B = 2,7 \text{ k}\Omega \text{ (valor comercial de resistência)}$$

A tensão v_s é ilustrada na fig. D.5:

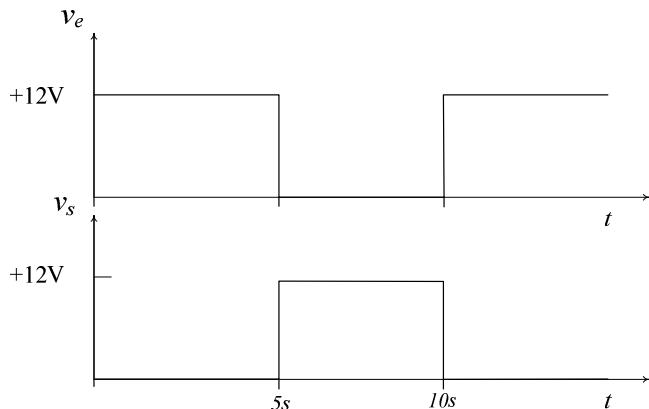


Fig. D.5 – Tensão aplicada no resistor de base e tensão resultante no coletor do transistor para o circuito da fig. D4.

Quando a tensão de entrada for 0 V, a chave fica ligada por 5 s e a carga consome 100 mA. A tensão na carga será aproximadamente 12 V (na prática existe uma pequena tensão entre coletor e emissor, especificada no catálogo do fabricante do transistor).

Exercício

D.1 - Deseja-se acionar o relé do circuito da fig. D.6⁵⁸ com um sinal de 5 V.

O relé deve operar com uma corrente de 150 mA. Será empregado o transistor NPN BC548 que tem um ganho mínimo de 110. Determinar R_B .

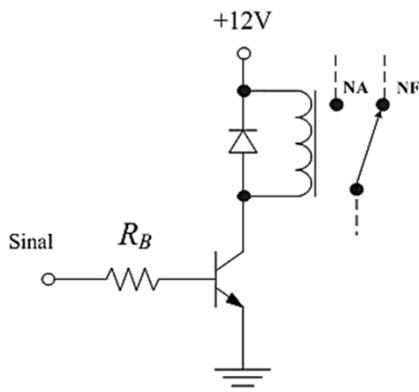


Fig. D.6 – Chave transistorizada para acionamento de um relé.

⁵⁸ Deve ser empregado um diodo, denominado diodo de roda livre, paralelo à bobina do relé, para evitar dano ao transistor durante o seu desligamento. Isso é necessário porque no corte da corrente de uma bobina, aparece uma força contra eletromotriz que induz uma tensão na bobina. Essa tensão possui polaridade oposta à de alimentação e sua magnitude pode ser elevada. Desta forma, o diodo de roda livre garante a desmagnetização gradual do indutor (dissipação da energia acumulada), grampeando a tensão induzida em cerca de 0,7 V até que a corrente no indutor se anule.

D.2 CIRCUITOS INTEGRADOS PARA O SUPRIMENTO DE CORRENTES

Dependendo do tamanho da placa de circuito impresso, muitas vezes é interessante empregar circuitos integrados dedicados para o suprimento de correntes, ou mesmo, para a simplificação do circuito. Por exemplo, o ULN2003 é utilizado para alimentar cargas que consomem até 500 mA, sendo acionado com tensões de 5 V e correntes de 1,35 mA; possui saídas em coletor aberto que suportam tensões de até 50 V e dispõem de diodos adequadamente conectados para permitir a conexão direta de cargas indutivas nas suas 7 saídas, como mostrado na fig. D.7. Na fig. D.8, é apresentado um circuito exemplo para o acionamento de um motor de passo bipolar e três motores DC, alimentados com 12 V.

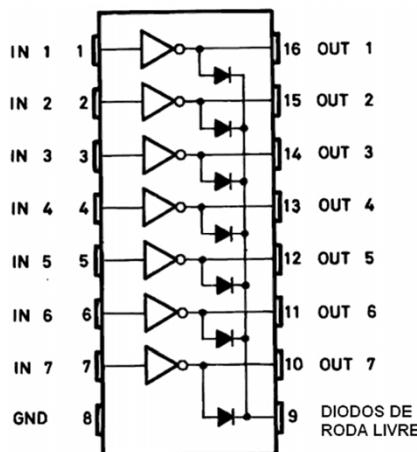


Fig. D.7 – Diagrama esquemático do ULN2003 (fonte ST microelectronics).

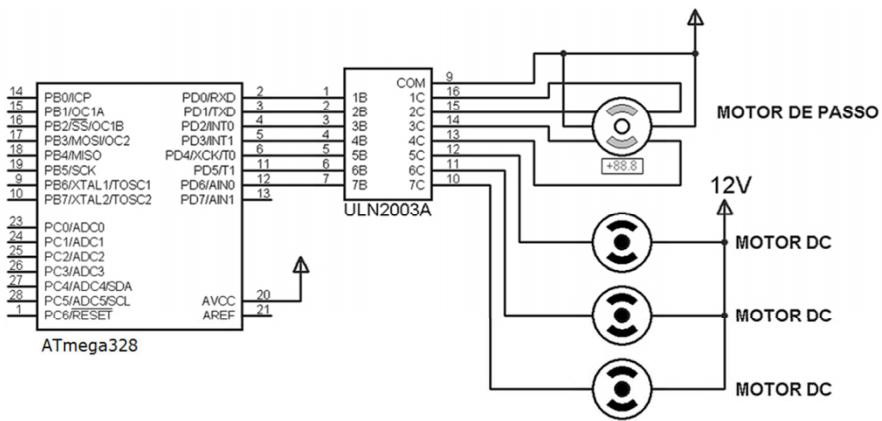


Fig. D.8 – Exemplo do emprego do ULN2003 para o acionamento de motores.

A família de CI's ULN possui outros componentes com diferentes características: o ULN2803, por exemplo, possui 8 saídas, sendo adequado para o trabalho com 1 byte de informação.

Quando a exigência de corrente não é grande, pode-se empregar circuitos digitais como o *buffer tri-state* 74244, que dependendo do fabricante, pode fornecer até 25 mA por pino.

Dependendo das exigências do projeto, outros CI's podem ser utilizados. O mercado disponibiliza uma infinidade dos chamados *drivers* ou *buffers* de corrente.

D.3 ACOPLADORES ÓPTICOS

Em muitas aplicações existe a necessidade de uma interface de isolamento entre o sistema digital de controle e o sistema a ser controlado (isolamento entre o sistema digital e o sistema analógico). Essa necessidade se deve ao ruído que pode ser induzido através das conexões elétricas entre os dois circuitos, prejudicando o funcionamento dos mesmos, ou da necessidade de isolamento elétrico entre circuitos com tensões diferentes. Esse isolamento pode ser feito com relés ou através de chaves ópticas especialmente projetadas (circuitos integrados especiais). A desvantagem dos relés está no tempo de fechamento dos seus contatos, no seu tamanho, na existência de partes mecânicas (que sofrem desgaste) e na geração de ruído elétrico. Os circuitos integrados são mais rápidos e menores; por outro lado, os relés trabalham com tensões e correntes superiores. Trabalhar com relés é relativamente fácil, entretanto, para trabalhar com circuitos integrados são necessários alguns cuidados especiais no dimensionamento dos componentes externos.

Para proporcionar o isolamento entre circuitos de corrente contínua podem ser empregados, por exemplo, os CIs 4N25 ou TIL111 (o diagrama esquemático e o encapsulamento do TIL11 é apresentado na fig. D.9). Esses CIs são compostos por um LED e um foto-transistor, quando o LED é acionado, o foto-transistor conduzirá. Desse modo, a corrente do LED deve ser suficiente para polarizar o foto-transistor. O projetista dimensiona o resistor para limitar a corrente do LED de acordo com a tensão de controle do componente.

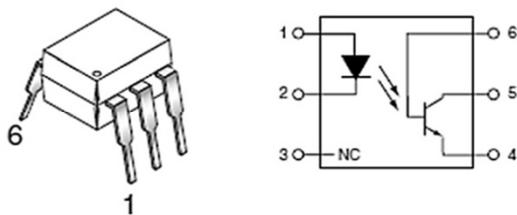


Fig. D.9 – Encapsulamento e diagrama esquemático de um acoplador óptico TIL111 (fonte Fairchild).

Quando se projeta um circuito acoplado opticamente, deve-se ter o cuidado de isolar fisicamente o terra da parte de controle do terra da parte controlada, conforme as necessidades do projeto.

Em corrente alternada, o acionamento de cargas por um circuito digital é bem mais delicado que o acionamento em corrente contínua devido às altas tensões envolvidas (da rede elétrica). Para tal, existem acopladores ópticos dedicados, como o MOC3041, visto na fig. D.10. O MOC3041 possui um LED que aciona um foto-triac, de acordo com o valor da corrente de carga. Além do MOC, pode ser necessário um triac no acionamento. Na fig. D.11, são apresentados circuitos exemplos para o acionamento de cargas em 115 VAC e 240 VAC, conforme o manual do fabricante.

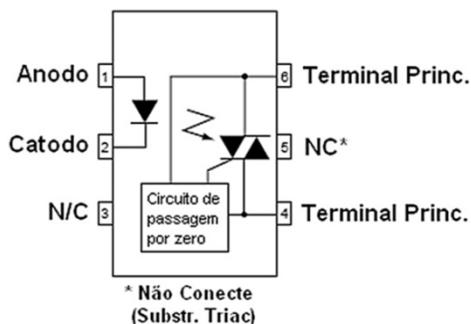
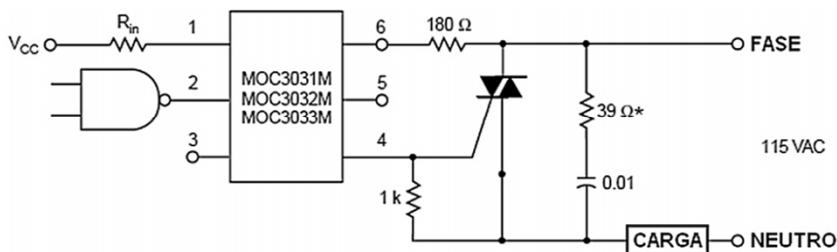
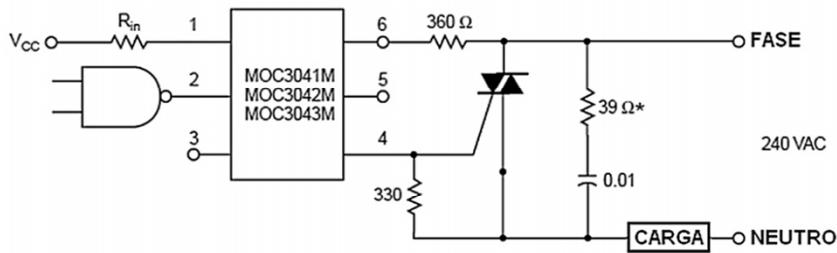


Fig. D.10 – Diagrama esquemático de um acoplador óptico MOC3041 (fonte Fairchild).



* Para cargas altamente indutivas (fator de pot. < 0,5), mudar este valor para 360 ohms.



* Para cargas altamente indutivas (fator de pot. < 0,5), mudar este valor para 360 ohms.

Fig. D.11 – Circuitos exemplos para o emprego do MOC3041 (fonte Fairchild).

E. PROJETO DE PLACAS DE CIRCUITO IMPRESSO - BÁSICO

Para garantir um perfeito funcionamento e durabilidade de um circuito eletrônico, seu projeto deve contemplar um bom desenho da placa de circuito impresso (PCI). O desenho, a espessura de trilhas e a disposição dos componentes na placa de circuito devem seguir regras para assegurar que o diagrama esquemático do circuito se comporte como desejado e que as normas de compatibilidade eletromagnética⁵⁹ sejam respeitadas.

O desenho de uma PCI é baseado no diagrama esquemático do circuito, o qual é feito em um software adequado. Dependendo da complexidade do projeto, o circuito necessitará ser simulado e montado para garantir que realmente funcionará. Após o projeto do circuito, é hora de desenhar a PCI. Os programas profissionais de projeto permitem que as conexões do diagrama esquemático sejam exportadas para o *software* de desenho (no caso do Proteus, o software ARES). Neste ponto, entram as habilidades de desenho e conhecimento técnico do desenhista ou ‘layoutista’.

E.1 UNIDADE IMPERIAL E MÉTRICA

A unidade mais empregada no desenho de PCIs ainda é a polegada. Assim, a maioria dos componentes eletrônicos apresenta suas dimensões nessa unidade, bem como as trilhas e vias nos *softwares* de desenho. A polegada é chamada de unidade imperial, enquanto o milímetro é a unidade métrica. Na tab. E.1, é apresentada a relação entre essas unidades e os nomes usuais encontrados.

⁵⁹ O circuito projetado não pode produzir interferência em outros equipamentos, nem sofrer interferência externa, sendo que os níveis de ruído produzidos devem respeitar normas específicas.

Tab. E.1 – Relação entre a unidade imperial e a métrica.

Imperial (polegada)	Métrica (mm)	Nome usual
1	25,4 (2,54cm)	1 inch pitch ou pitch
0,2	5,08	0,2 inch pitch ou pitch
0,1	2,54	100 mils ou 100 th (thou)
0,05	1,27	50 mils ou 50 th (thou)
0,01	0,25	10 mils ou 10 th (thou)

O valor de referência no desenho dos encapsulamentos dos componentes (distância entre pinos) é de 100 mils, seguido de suas frações, como 50, 75 e 200 mils, por exemplo. Para os desenhos, a regra geral é sempre usar a unidade imperial.

E.2 ENCAPSULAMENTOS

Os invólucros dos componentes eletrônicos ou encapsulamentos estão disponíveis em bibliotecas fornecidas com o *software* de desenho. Todavia, todo desenhista acaba criando seus próprios componentes, pois muitos não existem e outros necessitam ser alterados para se adequar às necessidades do projeto. Existem dois tipos de componentes eletrônicos, os que utilizam terminais que atravessam a placa de circuito impresso (PTH - *Plated Through-Hole*, inserido através de furo) e os componentes que são soldados na superfície da placa (SMD – *Surface Mount Device*, dispositivos de montagem na superfície). Para poder definir que tipo de componente será empregado na montagem da placa, todo desenhista deve se familiarizar com o nome dado aos diversos encapsulamentos. Esses são desenhados conforme o espaçamento entre os pinos do componente, sendo representados por *pads*. Para os componentes PTH, os *pads* indicam o cobre em torno do pino do componente e o diâmetro do furo por onde o pino será inserido. Para os componentes SMD, os *pads* são áreas de cobre que ficarão na superfície da placa para a soldagem dos terminais do componente. O encapsulamento também apresenta o desenho que será

impresso sobre a PCI, o que facilita a identificação do tipo de componente eletrônico, fornecendo referência para a sua soldagem e orientação.

E.2.1 COMPONENTES PTH

Na fig. E.1, são apresentados alguns exemplos de encapsulamentos para capacitores eletrolíticos e cerâmicos, com seus respectivos nomes. Observar que as distâncias entre os pinos são dadas em mils (th).

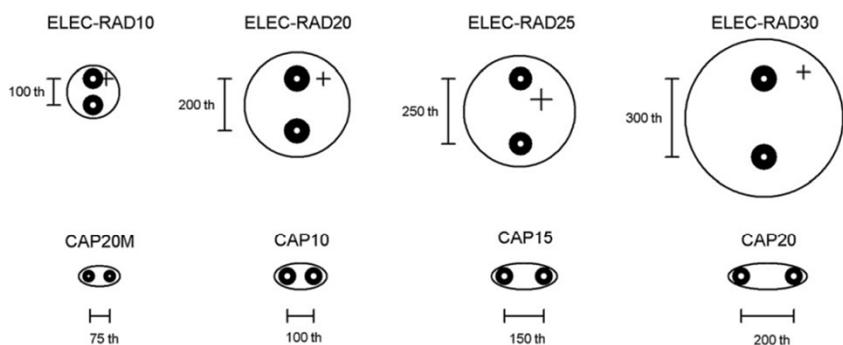


Fig. E.1. - Encapsulamento para capacitores eletrolíticos e cerâmicos PTH.

Para os resistores, o tamanho do encapsulamento está geralmente associado com a potência que será dissipada pelo mesmo. Na fig. E.2, são apresentados exemplos para resistores PTH, com seus respectivos nomes.

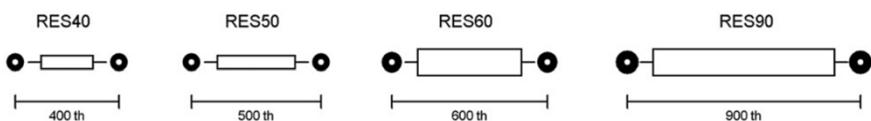


Fig. E.2. - Encapsulamento para resistores PTH.

Da mesma forma que para os resistores, o tamanho do encapsulamento dos diodos está relacionado com a capacidade de dissipação de potência (no caso dos diodos, também com a capacidade de

condução de corrente). Alguns exemplos são apresentados na fig. E.3. Nestes desenhos, o retângulo menor do encapsulamento indica o catodo do diodo.

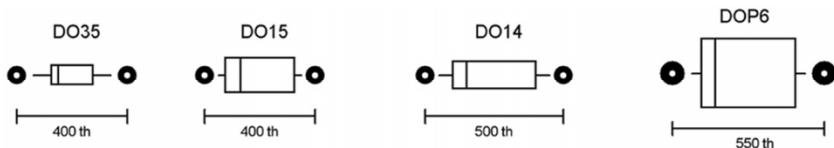


Fig. E.3. - Encapsulamento para diodos PTH.

Na fig. E.4, são apresentados alguns encapsulamentos para componentes eletrônicos de 3 terminais, tais como: transistores, triacs, tiristores, reguladores e sensores. A sequência dos pinos é indicada pelo desenho do encapsulamento.



Fig. E.4 – Encapsulamento para componentes eletrônicos de 3 terminais.

Uma grande parcela de circuitos integrados com mais de 3 pinos emprega o encapsulamento duplo em linha (DIL – *Dual In Line*, ou DIP – *Dual In line Package*), conforme exemplos da fig. E.5.

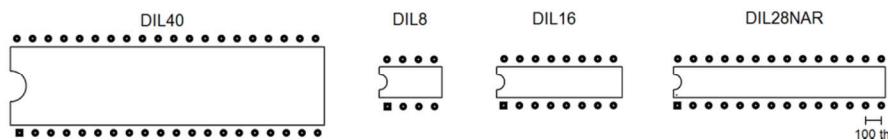


Fig. E.5 - Circuitos integrados com encapsulamento duplo em linha.

Na fig. E.6, são apresentadas as barras de pinos em linha simples e dupla para os encapsulamentos PTH.

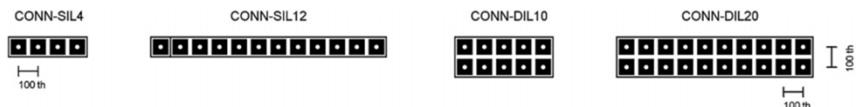


Fig. E.6 – Encapsulamento para barra de pinos.

E.2.2 COMPONENTES SMD

Os componentes de superfície são feitos para serem soldados na superfície das placas de circuito impresso, apresentando tamanho inferior aos componentes PTH de mesma funcionalidade. Assim, eles são adequados para a miniaturização de circuitos eletrônicos, permitindo a produção de PCIs extremamente pequenas. Entretanto, sua solda manual é complicada, exigindo ferros de solda com pontas bem finas ou equipamentos especiais. Quando a placa de circuito impresso empregar componentes SMDs e a solda dos mesmos for manual (com ferro de solda), é importante aumentar o tamanho usual dos *pads* dos encapsulamentos, facilitando, assim, a soldagem.

Na fig. E.7, são apresentados exemplos de encapsulamentos usuais para componentes passivos, a maioria para capacitores e resistores (0402 → 2512); um transistor (SOT23) e um diodo (DSM). O tamanho do componente tem relação com a potência que os mesmos podem dissipar. Quanto menor o componente, mais difícil é a sua solda manual.

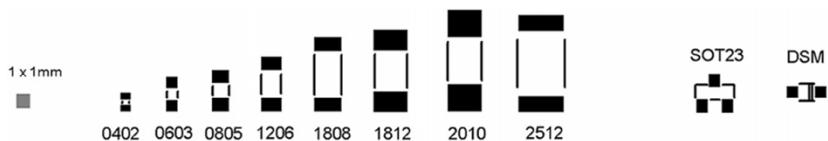


Fig. E.7 – Encapsulamentos SMD: componentes passivos 0402 – 2512, SOT23 para transistor e DSM para diodo.

Os circuitos integrados SMD possuem uma infinidade de encapsulamentos, incluindo componentes com pinos na parte de baixo, formando uma matriz de pontos (BGA - *Ball Grid Array*, soldado somente

com máquina específica). Na fig. E.8, são apresentados dois encapsulamentos comuns para circuitos integrados: o TQFP (*Thin Quad Flat Pack*), empregado, por exemplo, nos microcontroladores ATmega32 e o SOIC (*Small-Outline Integrated Circuit*), encapsulamento empregado em uma infinidade de circuitos integrados, tais como: ADCs, DACs, amplificadores operacionais e circuitos de lógica digital.

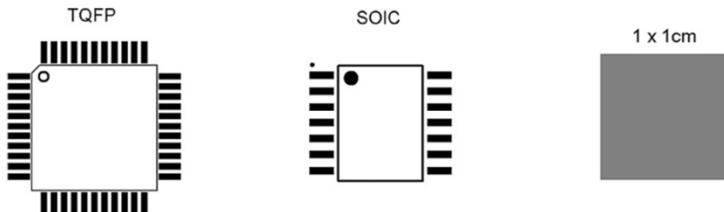


Fig. E.8 – Exemplos de encapsulamentos de circuitos integrados SMD.

E.3 PADS E VIAS (PTH)

Como comentado anteriormente, o *pad* descreve o cobre em torno do pino de um componente (neste caso, PTH). É o local onde será realizada a solda do componente e por onde o pino será inserido. Portanto, o *pad* possui um furo interno. Quando se desenha uma PCI é necessário o cuidado de não se errar as dimensões dos *pads*, senão se tornará muito difícil atravessar os pinos dos componentes pelos furos. Existem basicamente três tipos de *pads*: os circulares (DILCC), os quadrados (DILSQ) e os retangulares (STDDIL). Na fig. E.9, são apresentadas algumas configurações de *pads* circulares e quadrados, o primeiro número descreve a dimensão externa do *pad*, o segundo, o diâmetro do furo (em mils).

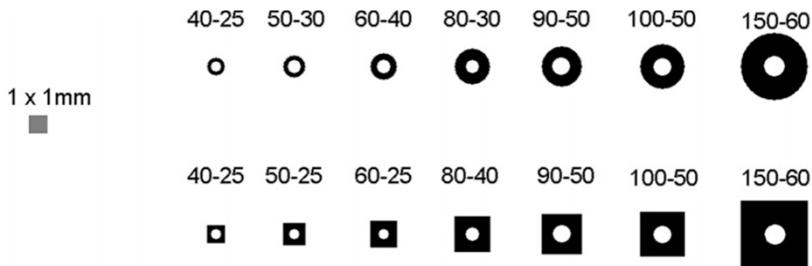


Fig. E.9 - *Pads PTH, X-Y* (*X* = diâmetro ou largura externa, *Y* = diâmetro do furo interno, em mils).

As vias são similares aos *pads*, entretanto, são empregadas para realizar conexões elétricas em placas com mais de uma superfície de cobre. Portanto, servem para ligar eletricamente as trilhas da parte superior da placa às trilhas da parte inferior, ou de camadas intermediárias, se existirem.

Dimensões usuais de vias são: 40-15 th, 50-20 th, 60-20 th, 70-20 th, 80-20 th. Como não será transpassada por pino, o diâmetro interno da via é menor que o dos *pads*. É boa prática manter o diâmetro das vias igual ao dobro da largura da trilha que conecta. Em frequências elevadas vias maiores possuem maior indutância, por isso, devem ser evitadas.

E.4 TRILHAS

Trilha é o caminho de cobre sobre a superfície da placa do circuito impresso por onde as correntes elétricas do circuito fluirão. Quanto menor o comprimento da trilha, menor será sua resistência, capacitância e indutância intrínsecas. Quanto mais espessa, maior sua resistência mecânica e capacidade elétrica. Na fig. E.10, são apresentadas as trilhas usuais empregadas em PCIs. Quando menor o tamanho da trilha, mais difícil é o seu processo de fabricação.

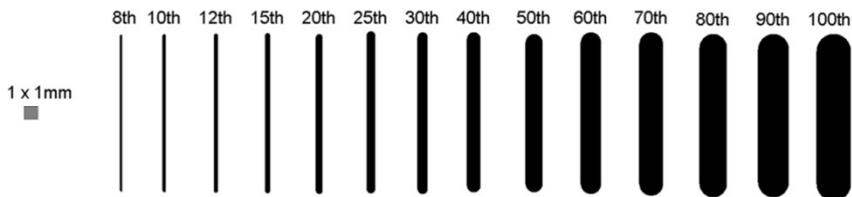


Fig. E.10 - Trilhas e suas larguras.

A trilha deve ter sua largura mínima determinada pela corrente que a irá percorrer. Essa largura depende da espessura do cobre da placa. Como as unidades de desenho ainda são antigas, emprega-se a onça/pé² (oz) para descrever o peso do cobre por unidade de superfície. O padrão é 1 oz, mas existem 0,5, 2 e 4 oz. Na tab. E.2, é apresentada a largura mínima de trilha que deve ser empregada para uma determinada corrente. É bom manter um coeficiente de segurança no dimensionamento das trilhas, lembrando que o aumento de temperatura demanda uma trilha mais larga do que uma empregada em uma temperatura menor. A resistência elétrica da trilha pode ser calculada empregando-se o volume do cobre da trilha e a condutividade elétrica do cobre, como explanado em livros de física do ensino médio.

Tab. E.2 – Largura mínima da trilha de acordo com a corrente que deverá suportar.

Largura da trilha (cobre)	Corrente [A]	Corrente [A]
	1 oz*	2 oz
5 mils	0,5	0,7
10 mils	0,8	1,4
20 mils	1,4	2,2
30 mils	1,9	3,0
50 mils	2,5	4,0
100 mils	4,0	7,0

* oz = 1 onça/pé² = 30 mg/cm².

Outra questão importante no desenho das trilhas é a distância entre trilhas adjacentes ou entre pontos eletricamente distintos. A distância é

empregada de acordo com a diferença de tensão entre os pontos em questão. Na tab. E.3, é apresentada a distância mínima entre trilhas de acordo com a diferença de potenciais elétricos a que possam estar submetidas; a mesma distância vale para os *pads*, vias, etc. Geralmente, é recomendado aumentar essas distâncias, melhorando o isolamento entre as trilhas.

Tab. E.3 – Distância mínima entre trilhas de acordo com a diferença de potencial a que estarão sujeitas.

Tensão (DC ou AC de pico)	Distância entre trilhas	
0-30 V	0,1 mm	8 mils
31-50 V	0,6 mm	25 mils
51-100 V	1,5 mm	60 mils
101-170 V	3,2 mm	150 mils
171-250 V	6,4 mm	300 mils
251-500 V	12,5 mm	500 mils

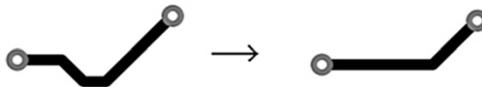
E.5 REGRAS BÁSICAS DE DESENHO

A seguir, são apresentadas inúmeras regras para um bom desenho da placa de circuito impresso. As regras ou são de ordem estética, ou de compatibilidade eletromagnética. É bom compreender que toda regra pode ter exceções, desde que se saiba o que se está fazendo. Com a prática, as técnicas de desenho vão sendo aperfeiçoadas e os desenhos terão uma melhor aparência e organização, apresentando, também, robustez elétrica e mecânica.

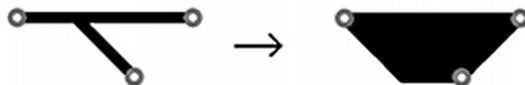
Regras para um bom desenho e qualidade de uma PCI:

- Organizar os componentes na PCI sempre que possível empregando simetria. A disposição dos componentes deve ser bem pensada, de acordo com as questões elétricas, de fixação da placa após montada e conforme será realizada a solda (automatizada ou manual).

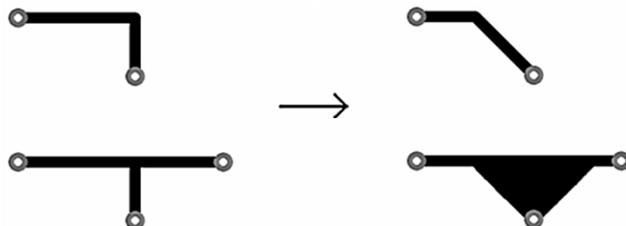
- Dispor os componentes com ligações entre si as mais próximas possíveis, ou seja, utilizar conexões curtas.



- O dimensionamento das trilhas, vias e *pads* devem ser adequados aos componentes, tensões e correntes que estarão presentes no circuito.
- Empregar polígonos nas trilhas de potência, sempre que possível; nas de sinal empregar polígonos de pequenos tamanhos.



- Empregar trilhas de maior espessura sempre que possível (a espessura de uma única trilha pode diminuir e aumentar dependendo do desenho).
- Nos planos de alimentação, utilizar a maior área de cobre possível.
- No desenho das trilhas, evitar mudanças de direção com ângulos de 90° ou outros que não 45°. O padrão é 45° e não deve ser alterado. Esse ângulo melhora o desenho da placa e suas características elétricas. Curvas são possíveis, mas não são comuns.



- Separar espacialmente circuitos de potência de circuitos digitais.

- Separar terra digital de terra analógico. A interconexão entre esses terras pode ser realizada em um ou mais pontos. Depende das questões de compatibilidade eletromagnética.
- Se possível, os circuitos digitais de maior frequência devem ficar localizados no centro da placa.
- Para placas com mais de uma superfície de cobre, criar planos independentes de terra e alimentação (VCC). Quando a placa possuir apenas duas faces, pode ser criado apenas um plano de terra.
- As trilhas de alimentação devem ser bem dispostas no desenho, se forem críticas, devem ser as primeiras a ser desenhadas. Procurar distribuir homogeneamente as correntes elétricas ao longo do circuito de cobre. As trilhas de terra e alimentação devem ser as mais largas possíveis e, ainda, devem estar próximas, evitando laços de corrente.
- Evitar trilhas de potência ao longo da extensão da placa. Se a placa tiver uma entrada de tensão alternada e uma saída de baixa tensão, uma deve estar de um lado da placa e a outra, do outro.
- Evitar a proximidade entre trilhas de potência e sinal e, mesmo, de alta potência e baixa potência. Caso o cruzamento das vias não possa ser evitado, em placas com mais de uma face de cobre, deve-se cruzar perpendicularmente vias de sinal e de potência. Regra geral: vias de sinal e potência devem estar afastadas.
- Aumentar a espessura do cobre próximo a terminais, *pads*, etc, sempre mantendo a estética. Isso melhora as características elétricas da trilha com o ponto que ela faz contato e aumenta a robustez mecânica para retrabalhos de soldagem.
- Não deixar planos de cobre isolados, sem conexão com o terra do circuito.
- Aterrkar a carcaça metálica de componentes que as tenham, tais como: cristais osciladores, dissipadores e blindagens.

- Em placas de face simples usar, preferencialmente, ligações com fios externos (*jumpers*) com orientação vertical e horizontal. Jamais fazer estas ligações entre pinos de componentes, é necessário colocar *pads* para isso.
- Capacitores de filtro para ruídos de alta frequência, os chamados capacitores de desacoplamento (geralmente de 100 nF), só tem utilidade quando colocados próximo ao circuito integrado que devem proteger.

Se o circuito trabalhar em frequências elevadas, onde o comprimento das trilhas é próximo ao comprimento de onda dos sinais empregados, muito das considerações acima terão que ser avaliadas. Questões de compatibilidade eletromagnética (emitir ruído elétrico ou sofrer interferência de ruídos externos) podem exigir desenhos fora do senso comum, com estética ruim, mas que apresentam as características elétricas desejadas.

Dependendo da complexidade do circuito, pode ser empregada uma placa de face simples (cobre em somente um dos lados), dupla face (cobre em ambos os lados) ou com N camadas, formando um sanduíche. Quanto mais simples a placa, menor é o seu custo. O viés da placa de face simples é a quantidade de *jumpers* que a mesma pode exigir e a sua menor robustez elétrica. Tudo vai depender do custo benefício do projeto.

As placas podem ser feitas de fenolite (um tipo de papel impregnado com resina) ou fibra de vidro, com espessuras de 1 mm, 1,6 mm (padrão), 1,8 mm e 2,5 mm. As melhores são as de fibra de vidro e, obviamente, as mais caras. Existem outros materiais, como o composite, que está entre a fibra de vidro e o fenolite.

E.6 PASSOS PARA O DESENHO DA PCI

1. Desenhar o diagrama esquemático do circuito tendo a PCI em mente. Testar e ter certeza da funcionalidade do circuito. Nas figs. E.11a e E.11b, são apresentados exemplos de desenho de um mesmo diagrama esquemático: um ruim e outro bom. Em um diagrama esquemático, as linhas de conexão devem apresentar simetria, organização e serem capazes de expressar com clareza as ligações elétricas.

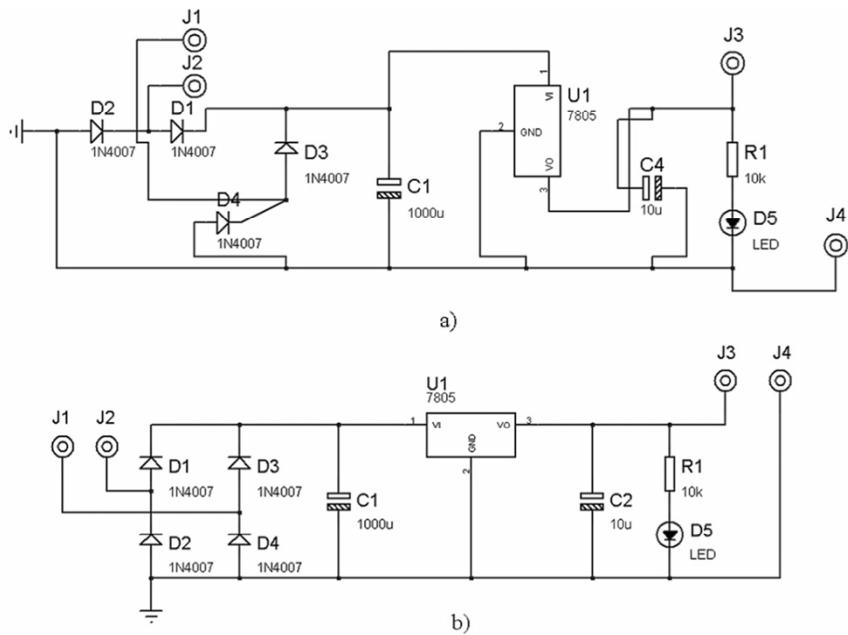


Fig. E.11 – Diagrama esquemático de circuitos eletrônicos: a) desenho confuso (exagerado para exemplo) e b) desenho organizado.

2. Transferir as ligações elétricas, o chamado *netlist*, para o software de confecção da PCI. Essa etapa é apresentada na fig. E.12. Observar a organização dada aos componentes, conforme o diagrama da fig. E.11b.

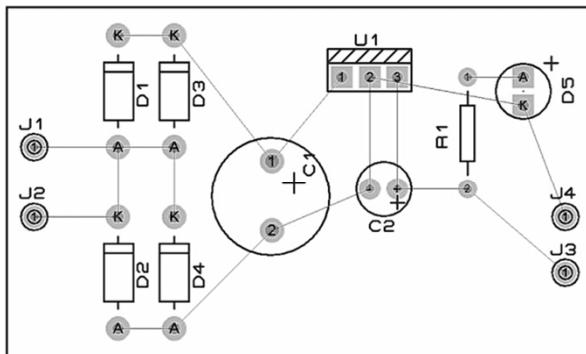


Fig. E.12 – Componentes eletrônicos e suas ligações elétricas (*netlist*) antes do desenho das trilhas.

3. Organizar os componentes na placa - etapa fundamental. Deve-se gastar um bom tempo dispondo da melhor forma os componentes eletrônicos na placa, pois isso facilitará e garantirá uma boa qualidade no desenho final. Na fig. E.12, é apresentada uma boa organização. A organização, também depende do local onde a placa será fixada. Dependendo do software empregado é possível visualizar tridimensionalmente como ficará a montagem da placa (fig. E.13).



Fig. E.13 – Imagem tridimensional da disposição dos componentes eletrônicos na PCI.

4. Utilizar a ferramenta do software de desenho para a geração automática das trilhas e conexões do circuito. Após isso, desfazer as trilhas e voltar ao passo 3, até se obter uma melhor organização das ligações elétricas.

5. Após a geração automática das trilhas, refazer manualmente trilha por trilha, de acordo com as regras de desenho. Um bom desenho é feito manualmente, jamais de forma automática. Nas figs. E.14a e E.14b, são apresentados um desenho ruim e um bom desenho da PCI para o circuito da fig. E.11.

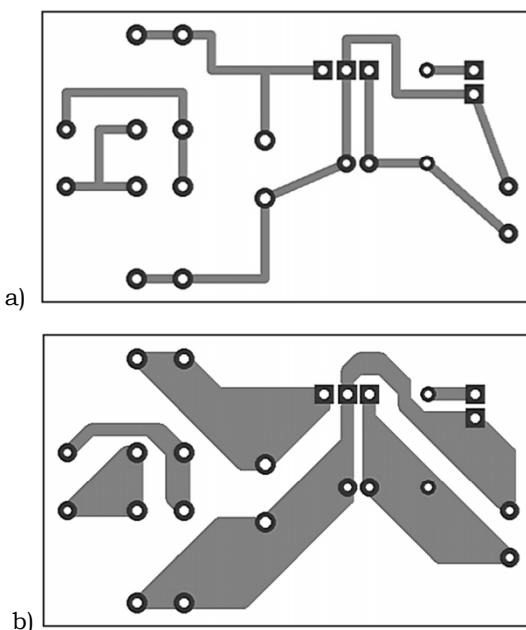


Fig. E.14 – a) PCI com um desenho ruim, b) PCI com um bom desenho.

Na fig. E.14a, percebe-se o descuido com os ângulos das trilhas, diferentes de 45°, o emprego de uma mínima área de cobre e conexões mal desenhadas. Já na fig. E.14b, é utilizada uma maior área de cobre, com o emprego de polígonos e trilhas largas; as conexões são robustas e eletricamente eficientes.

6. Depois das trilhas desenhadas, colocam-se os planos de terra e alimentação. Isso permite uma melhor distribuição das correntes ao longo da placa, como é apresentado na fig. E.15.

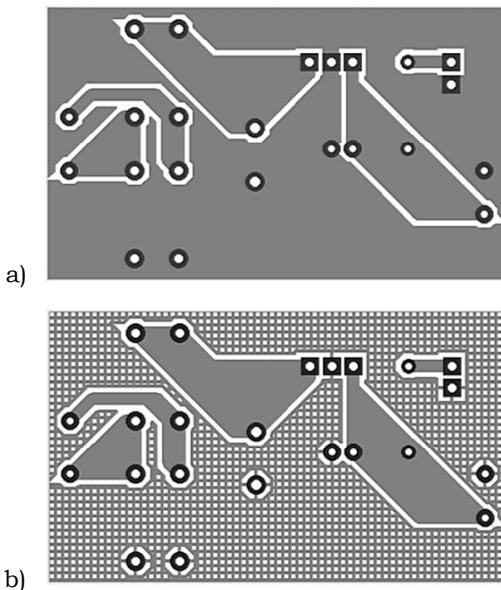


Fig. E.15 – a) PCI com um plano de terra maciço e b) PCI com plano de terra quadriculado e *relieve pins*.

O plano de terra ou alimentação pode ser maciço (cobre contínuo) ou quadriculado e possuir conexões nos *pads* para facilitar a soldagem (*relieve pins*), as quais são importantes quando a placa será soldada automaticamente. Existe certa controvérsia sobre qual forma de plano é melhor: se maciço ou quadriculado. Sob transformadores, o plano quadriculado deve ter melhor resultado, pois diminui as correntes parasitas (correntes de Foucault).

7. Detalhes finais: revisão das regras empregadas e desenho da parte impressa da placa (*silkscreen*). A parte impressa é muito importante para a montagem e uso da PCI, pois contém dados, como: a numeração dos componentes, nomes de pinos e outras funcionalidades. Deve conter o nome ou número da placa, data e versão do projeto, bem como a assinatura do desenhista. Essas informações podem ser feitas no cobre da placa. Uma placa que se preza deve ser assinada!

8. Conferir calmamente todo o trabalho, incluindo a revisão das conexões elétricas entre os componentes eletrônicos.

Nas PCIs apresentadas, para uma melhor visualização, a superfície do desenho não foi espelhada. A superfície de cobre na prática estará na parte inferior da placa e os componentes serão colocados na parte superior, lembrando que essas são placas de face simples.

A seguir, são apresentadas figuras para o exemplo de um projeto real de uma placa de circuito impresso de dupla face (Tri Motor *Shield* – para o Arduino, fig. 9.17 do capítulo 9), que emprega componentes PTHs. Na fig. E.16, é apresentada a parte inferior da placa (*bottom*) onde está localizado um dos plano de terra e, na fig. E.17, a parte superior da placa (*top*) onde está localizado o outro plano de terra (é possível utilizar somente um plano de terra, ou um plano de terra e outro de alimentação, formando um capacitor com a placa). É importante analisar os desenhos, observando a simetria das trilhas e as regras que foram empregadas. Na fig. E.18, é apresentado o *silkscreen*, que foi cuidadosamente desenhado para facilitar o uso da PCI. Nesse projeto não foram previstos furos para fixação. Quando houver necessidade, o projetos deve dispor de espaços para furação, cuidando, nesses casos, de manter as trilhas suficientemente afastadas dos furos.

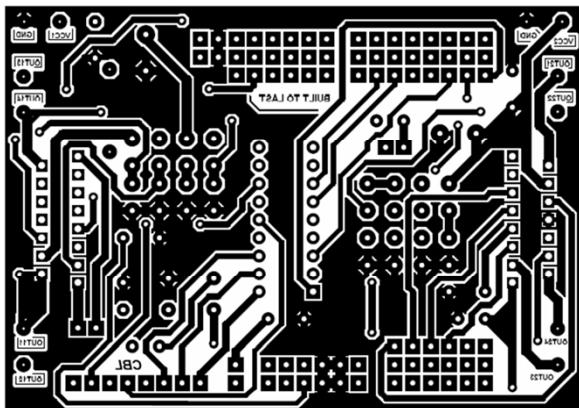


Fig. E.16 – Desenho da PCI da parte inferior de uma placa de dupla face (Tri Motor Shield para o Arduino).

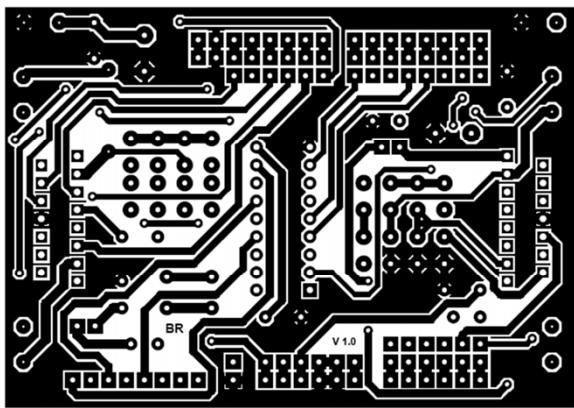


Fig. E.17 – Desenho da PCI da parte superior de uma placa de dupla face (Tri Motor Shield para o Arduino).

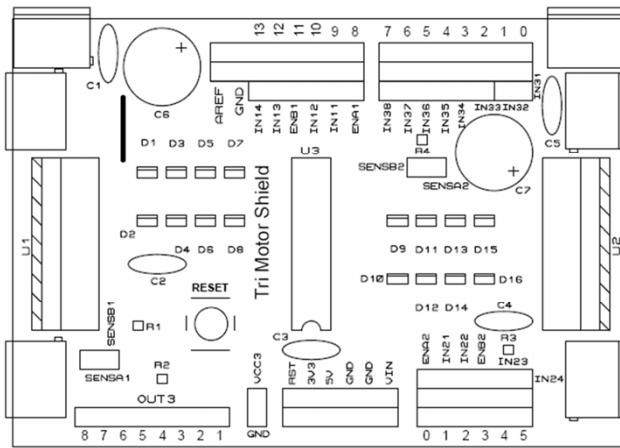


Fig. E.18 – Desenho da parte a ser impressa sobre a parte superior da PCI (Tri Motor Shield para o Arduino).

E.7 CONCLUSÕES

As regras de desenho e estética da PCI devem ser seguidas sempre que possível, garantindo a qualidade da PCI.

Desenhar uma placa de circuito impresso exige muito mais que conhecimento técnico, é uma questão de arte. Bons desenhistas são raros e demandam muita prática para garantir excelência.

Para atender as exigências de compatibilidade eletromagnética pode ser necessário o desenho e teste de várias PCIs para um mesmo circuito. Projetos por tentativa e erro ainda são comuns. Para frequências elevadas, a disposição das trilhas e componentes se torna delicada, exigindo cuidados e o conhecimento de teorias complexas de eletromagnetismo.

A qualidade de uma placa é diretamente proporcional à prática, ao conhecimento do desenhista e ao tempo gasto no seu desenho. O desenho de uma boa PCI, com certeza, demanda muitas horas de trabalho.

F. TABELAS DE CONVERSÃO

Tab. F1: Código ASCII.

Caractere	Dec	Hex	Caractere	Dec	Hex
NUL (Null char.)	0	00	SP (Space)	32	20
SOH (Start of Header)	1	01	!	33	21
STX (Start of Text)	2	02	"	34	22
ETX (End of Text)	3	03	#	35	23
EOT (End of Transmission)	4	04	\$	36	24
ENQ (Enquiry)	5	05	%	37	25
ACK (Acknowledgment)	6	06	&	38	26
BEL (Bell)	7	07	'	39	27
BS (Backspace)	8	08	(40	28
HT (Horizontal Tab)	9	09)	41	29
LF (Line Feed)	10	0A	*	42	2A
VT (Vertical Tab)	11	0B	+	43	2B
FF (Form Feed)	12	0C	,	44	2C
CR (Carriage Return)	13	0D	-	45	2D
SO (Shift Out)	14	0E	.	46	2E
SI (Shift In)	15	0F	/	47	2F
DLE (Data Link Escape)	16	10	0	48	30
DC1 (XON)(Device Control 1)	17	11	1	49	31
DC2 (Device Control 2)	18	12	2	50	32
DC3 (XOFF)(Device Control 3)	19	13	3	51	33
DC4 (Device Control 4)	20	14	4	52	34
NAK (Neg. Acknowledgement)	21	15	5	53	35
SYN (Synchronous Idle)	22	16	6	54	36
ETB (End of Trans. Block)	23	17	7	55	37
CAN (Cancel)	24	18	8	56	38
EM (End of Medium)	25	19	9	57	39
SUB (Substitute)	26	1A	:	58	3A
ESC (Escape)	27	1B	;	59	3B
FS (File Separator)	28	1C	<	60	3C
GS (Group Separator)	29	1D	=	61	3D
RS (Request to Send – Rec. Sep.)	30	1E	>	62	3E
US (Unit Separator)	31	1F	?	63	3F

Caractere	Dec	Hex	Caractere	Dec	Hex
@	64	40	`	96	60
A	65	41	a	97	61
B	66	42	b	98	62
C	67	43	c	99	63
D	68	44	d	100	64
E	69	45	e	101	65
F	70	46	f	102	66
G	71	47	g	103	67
H	72	48	h	104	68
I	73	49	i	105	69
J	74	4A	j	106	6A
K	75	4B	k	107	6B
L	76	4C	l	108	6C
M	77	4D	m	109	6D
N	78	4E	n	110	6E
O	79	4F	o	111	6F
P	80	50	p	112	70
Q	81	51	q	113	71
R	82	52	r	114	72
S	83	53	s	115	73
T	84	54	t	116	74
U	85	55	u	117	75
V	86	56	v	118	76
W	87	57	w	119	77
X	88	58	x	120	78
Y	89	59	y	121	79
Z	90	5A	z	122	7A
[91	5B	{	123	7B
/	92	5C		124	7C
]	93	5D	}	125	7D
^	94	5E	~	126	7E
-	95	5F	DEL	127	7F

Tab. F2: Conversão Decimal – Hexadecimal – Binária.

Dec.	Hex.	Bin.	Dec.	Hex.	Bin.	Dec.	Hex.	Bin.
0	00	00000000	43	2B	00101011	86	56	01010110
1	01	00000001	44	2C	00101100	87	57	01010111
2	02	00000010	45	2D	00101101	88	58	01011000
3	03	00000011	46	2E	00101110	89	59	01011001
4	04	00000100	47	2F	00101111	90	5A	01011010
5	05	00000101	48	30	00110000	91	5B	01011011
6	06	00000110	49	31	00110001	92	5C	01011100
7	07	00000111	50	32	00110010	93	5D	01011101
8	08	00001000	51	33	00110011	94	5E	01011110
9	09	00001001	52	34	00110100	95	5F	01011111
10	0A	00001010	53	35	00110101	96	60	01100000
11	0B	00001011	54	36	00110110	97	61	01100001
12	0C	00001100	55	37	00110111	98	62	01100010
13	0D	00001101	56	38	00111000	99	63	01100011
14	0E	00001110	57	39	00111001	100	64	01100100
15	0F	00001111	58	3A	00111010	101	65	01100101
16	10	00010000	59	3B	00111011	102	66	01100110
17	11	00010001	60	3C	00111100	103	67	01100111
18	12	00010010	61	3D	00111101	104	68	01101000
19	13	00010011	62	3E	00111110	105	69	01101001
20	14	00010100	63	3F	00111111	106	6A	01101010
21	15	00010101	64	40	01000000	107	6B	01101011
22	16	00010110	65	41	01000001	108	6C	01101100
23	17	00010111	66	42	01000010	109	6D	01101101
24	18	00011000	67	43	01000011	110	6E	01101110
25	19	00011001	68	44	01000100	111	6F	01101111
26	1A	00011010	69	45	01000101	112	70	01110000
27	1B	00011011	70	46	01000110	113	71	01110001
28	1C	00011100	71	47	01000111	114	72	01110010
29	1D	00011101	72	48	01001000	115	73	01110011
30	1E	00011110	73	49	01001001	116	74	01110100
31	1F	00011111	74	4A	01001010	117	75	01110101
32	20	00100000	75	4B	01001011	118	76	01110110
33	21	00100001	76	4C	01001100	119	77	01110111
34	22	00100010	77	4D	01001101	120	78	01111000
35	23	00100011	78	4E	01001110	121	79	01111001
36	24	00100100	79	4F	01001111	122	7A	01111010
37	25	00100101	80	50	01010000	123	7B	01111011
38	26	00100110	81	51	01010001	124	7C	01111100
39	27	00100111	82	52	01010010	125	7D	01111101
40	28	00101000	83	53	01010011	126	7E	01111110
41	29	00101001	84	54	01010100	127	7F	01111111
42	2A	00101010	85	55	01010101			

Dec.	Hex.	Bin.	Dec.	Hex.	Bin.	Dec.	Hex.	Bin.
128	80	10000000	171	AB	10101011	214	D6	11010110
129	81	10000001	172	AC	10101100	215	D7	11010111
130	82	10000010	173	AD	10101101	216	D8	11011000
131	83	10000011	174	AE	10101110	217	D9	11011001
132	84	10000100	175	AF	10101111	218	DA	11011010
133	85	10000101	176	B0	10110000	219	DB	11011011
134	86	10000110	177	B1	10110001	220	DC	11011100
135	87	10000111	178	B2	10110010	221	DD	11011101
136	88	10001000	179	B3	10110011	222	DE	11011110
137	89	10001001	180	B4	10110100	223	DF	11011111
138	8A	10001010	181	B5	10110101	224	E0	11100000
139	8B	10001011	182	B6	10110110	225	E1	11100001
140	8C	10001100	183	B7	10110111	226	E2	11100010
141	8D	10001101	184	B8	10111000	227	E3	11100011
142	8E	10001110	185	B9	10111001	228	E4	11100100
143	8F	10001111	186	BA	10111010	229	E5	11100101
144	90	10010000	187	BB	10111011	230	E6	11100110
145	91	10010001	188	BC	10111100	231	E7	11100111
146	92	10010010	189	BD	10111101	232	E8	11101000
147	93	10010011	190	BE	10111110	233	E9	11101001
148	94	10010100	191	BF	10111111	234	EA	11101010
149	95	10010101	192	C0	11000000	235	EB	11101011
150	96	10010110	193	C1	11000001	236	EC	11101100
151	97	10010111	194	C2	11000010	237	ED	11101101
152	98	10011000	195	C3	11000011	238	EE	11101110
153	99	10011001	196	C4	11000100	239	EF	11101111
154	9A	10011010	197	C5	11000101	240	F0	11110000
155	9B	10011011	198	C6	11000110	241	F1	11110001
156	9C	10011100	199	C7	11000111	242	F2	11110010
157	9D	10011101	200	C8	11001000	243	F3	11110011
158	9E	10011110	201	C9	11001001	244	F4	11110100
159	9F	10011111	202	CA	11001010	245	F5	11110101
160	A0	10100000	203	CB	11001011	246	F6	11110110
161	A1	10100001	204	CC	11001100	247	F7	11110111
162	A2	10100010	205	CD	11001101	248	F8	11111000
163	A3	10100011	206	CE	11001110	249	F9	11111001
164	A4	10100100	207	CF	11001111	250	FA	11111010
165	A5	10100101	208	D0	11010000	251	FB	11111011
166	A6	10100110	209	D1	11010001	252	FC	11111100
167	A7	10100111	210	D2	11010010	253	FD	11111101
168	A8	10101000	211	D3	11010011	254	FE	11111110
169	A9	10101001	212	D4	11010100	255	FF	11111111
170	AA	10101010	213	D5	11010101			

G. REGISTRADORES DE I/O DO ATMEGA328

PORTs DE I/O

PORTB - PORT B Data Register

Bit	7	6	5	4	3	2	1	0
PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
Lê/Escrive	L/E							
Valor Inicial	0	0	0	0	0	0	0	0

DDRB - PORT B Data Direction Register

Bit	7	6	5	4	3	2	1	0
DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
Lê/Escrive	L/E							

PINB - PORT B Input Pins Address

Bit	7	6	5	4	3	2	1	0
PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
Lê/Escrive	L/E							

Obs.: para os outros PORTs tem-se PORTC, DDRC, PINC, PORTD, DDRD e PIND.

Configuração dos PORTs.

DDXn*	PORTXn	PUD (SFIOR)	I/O	Pull - up	Comentário
0	0	x	Entrada	Não	Alta impedância.
0	1	0	Entrada	Sim	O pino pode fornecer corrente.
0	1	1	Entrada	Não	PXn irá fornecer corrente se externamente for colocado em nível lógico 0.
1	0	x	Saída	Não	Saída, nível lógico baixo.
1	1	x	Saída	Não	Saída, nível lógico alto.

* X = B, C ou D; n = 0, 1, ... ou 7.

Obs.: o bit PUD desabilita o *pull-up* de todas as portas.

INTERRUPÇÕES EXTERNAS

EICRA - *Extern Interrupt Control Register A*

Bit	7	6	5	4	3	2	1	0
EICRA	-	-	-	-	ISC11	ISC10	ISC01	ISC00
Lê/Escrive	L	L	L	L	L/E	L/E	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

Bits de configuração da forma das interrupções nos pinos INT1 e INT0.

ISC11	ISC10	Pedido de Interrupção
0	0	Nível baixo em INT1
0	1	Mudança lógica em INT1
1	0	Borda de decida em INT1
1	1	Borda de subida em INT1

ISC01	ISC00	Pedido de Interrupção
0	0	Nível baixo em INT0
0	1	Mudança lógica em INT0
1	0	Borda de decida em INT0
1	1	Borda de subida em INT0

EIMSK - *External Interrupt Mask Register*

Bit	7	6	5	4	3	2	1	0
EIMSK	-	-	-	-	-	-	INT1	INT0
Lê/Escrive	L	L	L	L	L	L	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

Bit = 1, interrupção habilitada, bit = 0, interrupção desabilitada.

EIFR - *External Interrupt Flag Register*

Bit	7	6	5	4	3	2	1	0
EIFR	-	-	-	-	-	-	INTF1	INTFO
Lê/Escrive	L	L	L	L	L	L	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

Os bits sinalizam a solicitação de uma interrupção, são limpos pela escrita de 1 lógico na sua posição.

PCICR - Pin Change Interrupt Control Register

Bit	7	6	5	4	3	2	1	0
PCICR	-	-	-	-	-	PCIE2	PCIE1	PCIE0
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

PCIE0 = 1, habilita a interrupção por mudança nos pinos do PORTB (PCINT0:7); PCIE1 = 1, nos pinos do PORTC (PCINT8:14); PCIE2 = 1, nos pinos do PORTD (PCINT16:23).

PCIFR - Pin Change Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
PCIFR	-	-	-	-	-	PCIF2	PCIF1	PCIF0
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

Os bits sinalizam a solicitação de uma interrupção em um dos pinos do PORT: PCIF0 para o PORTB, PCIF1 para o PORTC e PCIF2 para o PORTD; são limpos pela escrita de 1 lógico na sua posição.

PCMSK0 - Pin Change Mask Register 0

Bit	7	6	5	4	3	2	1	0
PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
Lê/Escrive	L/E							
Valor Inicial	0	0	0	0	0	0	0	0

PCMSK1 - Pin Change Mask Register 1

Bit	7	6	5	4	3	2	1	0
PCMSK1	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

PCMSK2 - Pin Change Mask Register 2

Bit	7	6	5	4	3	2	1	0
PCMSK2	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
Lê/Escrive	L/E							
Valor Inicial	0	0	0	0	0	0	0	0

Bit = 1, interrupção habilitada no pino, desde que a interrupção esteja habilitada no PORT (ver PCICR).

T/C0

TIMSK0 - Timer/Counter 0 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
TIMSK0	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit = 1 – interrupção habilitada; bit = 0, interrupção desabilitada.

TOIE0 – interrupção por estouro de contagem.

OCIE0A, OCIE0B – interrupções por coincidência de comparação.

TIFR0 - Timer/Counter 0 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
TIMSK0	-	-	-	-	-	OCF0B	OCF0A	TOV0
Lê/Escrive	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Os bits sinalizam a solicitação de uma interrupção, são limpos pela escrita de 1 lógico na sua posição.

TCCR0A - Timer/Counter 0 Control Register A

Bit	7	6	5	4	3	2	1	0
TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00
Lê/Escrive	L/E	L/E	L/E	L/E	L	L	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

TCCR0B – Timer/Counter 0 Control Register B

Bit	7	6	5	4	3	2	1	0
TCCR0B	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00
Lê/Escrive	E	E	L	L	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

WGM02:0 – configuração do modo de operação do TC0.

COM0A1:0 e COM0B1:0 – configuram a ação nas saídas OC0A/OC0B de acordo com o modo escolhido.

FOC0A:B - *Force Output Compare*. Colocados em 1, uma comparação é forçada no módulo gerador de onda. As saídas são alteradas de acordo com os bits COM0A1:0 e COM0B1:0. Em zero nos modos PWM.

CS02:0 – seleção do *prescaler*.

TCNT0 – Timer/Counter 0 Register

Registrador de 8 bits onde é realizada a contagem do TC0, pode ser lido ou escrito a qualquer tempo.

OCR0A – Output Compare 0 Register A e OCR0B – Output Compare 0 Register B

Registradores de Comparação de 8 bits

Seleção do *clock* para o TC0

CS02	CS01	CS00	Descrição
0	0	0	TC0 parado.
0	0	1	CLK
0	1	0	CLK/8
0	1	1	CLK/64
1	0	0	CLK/256
1	0	1	CLK/1024
1	1	0	<i>Clock</i> externo no pino T0. Borda de descida.
1	1	1	<i>Clock</i> externo no pino T0. Borda de subida.

Frequência em OC0A [Hz]

Modo CTC

$$f_{OC0A} = \frac{f_{osc}}{2N(1 + OCR0A)}$$

Modo PWM rápido

$$f_{OC0A_PWM} = \frac{f_{osc}}{N(1 + TOP)}$$

Modo PWM com fase corrigida

$$f_{OC0A_PWM} = \frac{f_{osc}}{2N(1 + TOP)}$$

Configuração do modo de operação do TC0.

Modo	WGM02	WGM01	WGM00	Modo de Operação	TOP	Atualização de OCR0 no valor:	Sinalização do bit TOV0 no valor:
0	0	0	0	Normal	0xFF	Imediata	0xFF
1	0	0	1	PWM com fase corrigida	0xFF	0xFF	0x00
2	0	1	0	CTC	OCR0A	Imediata	0xFF
3	0	1	1	PWM rápido	0xFF	0x00	0xFF
4	1	0	0	reservado	-	-	-
5	1	0	1	PWM com fase corrigida	OCR0A	OCR0A	0x00
6	1	1	0	reservado	-	-	-
7	1	1	1	PWM rápido	OCR0A	0x00	OCR0A

Ação nas saídas OC0A/OC0B.

MODO NÃO PWM			MODO PWM RÁPIDO			MODO PWM FASE CORRIGIDA		
COM0A1 COM0B1	COM0A0 COM0B0	DESCRIÇÃO	COM0A1 COM0B1	COM0A0 COM0B0	DESCRIÇÃO	COM0A1 COM0B1	COM0A0 COM0B0	DESCRIÇÃO
0	0	OC0A/OC0B desconectado	0	0	OC0A/OC0B desconectado	0	0	OC0A/OC0B desconectado
0	1	Inverte OC0A/OC0B na comparação	0	1	Inverte OC0A na comparação*	0	1	Inverte OC0A na comparação*
1	0	OC0A/OC0B em zero na comparação	1	0	OC0A/OC0B = 0 na comparação, em 1 no final do período	1	0	OC0A/OC0B = 0 na comparação na contagem crescente, OC0A/OC0B = 1 na comparação na contagem decrescente.
1	1	OC0A/OC0B em alto na comparação	1	1	OC0A/OC0B = 1 na comparação, em 0 no final do período	1	1	OC0A/OC0B = 1 na comparação na contagem crescente, OC0A/OC0B=0 na comparação na contagem decrescente.

* Válido para WGM02 = 1, se WGM02 = 0: OC0A desconectado. COM0B1:0 = 1 é reservado.

T/C2

TIMSK2 – Timer/Counter 2 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
TIMSK2	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit = 1 – interrupção habilitada; bit = 0, interrupção desabilitada.

TOIE2 – interrupção por estouro de contagem.

OCIE2A, OCIE2B – interrupções por coincidência de comparação.

TIFR2 – Timer/Counter 2 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
TIFR2	-	-	-	-	-	OCF2B	OCF2A	TOV2
Lê/Escrive	L	L	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Os bits sinalizam a solicitação de uma interrupção, são limpos pela escrita de 1 lógico na sua posição.

TCCR2A - Timer/Counter 2 Control Register A

Bit	7	6	5	4	3	2	1	0
	TCCR2A							
Lê/Escr.	L/E	L/E	L/E	L/E	L	L	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

TCCR2B – Timer/Counter 2 Control Register B

Bit	7	6	5	4	3	2	1	0
	TCCR2B							
Lê/Escr.	E	E	L	L	L/E	L/E	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

WGM22:0 – configuração do modo de operação do TC2.

COM2A1:0 e COM2B1:0 – configuram a ação nas saídas OC2A/OC2B de acordo com o modo escolhido.

FOC2A:B - *Force Output Compare*. Colocados em 1, uma comparação é forçada no módulo gerador de onda. As saídas são alteradas de acordo com os bits COM2A1:0 e COM2B1:0. Em zero nos modos PWM.
CS22:0 – seleção do prescaler.

TCNT2 – Timer/Counter 2 Register

Registrador de 8 bits onde é realizada a contagem do TC2, pode ser lido ou escrito a qualquer tempo.

OCR2A – Output Compare 2 Register A e OCR2B – Output Compare 2 Register B

Registradores de Comparação de 8 bits.

ASSR – Asynchronous Status Register

Bit	7	6	5	4	3	2	1	0
	ASSR							
Lê/Escrve	-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB
Valor Inicial	L	L	L	L	L	L/E	L/E	L/E

AS2 = 1: um cristal externo de 32,768 kHz nos pinos TOSC1 e TOSC2 fornece o sinal de *clock* ao TC2.
EXCLK = 1 e **AS2 = 1**, um clock externo é entrada para o pino TOSC1.

TCNT2UB, **OCR2AUB**, **OCR2BUB**, **TCR2AUB** e **TCR2BUB** sinalizam a escrita nos registradores TCNT2, OCR2B, OCR2A, TCCR2A e TCCR2B, respectivamente.

Seleção do *clock* para o TC2

Frequência em OC2A/OC2B [Hz]

CS22	CS21	CS20	Descrição
0	0	0	TC2 parado.
0	0	1	CLK
0	1	0	CLK/8
0	1	1	CLK/32
1	0	0	CLK/64
1	0	1	CLK/128
1	1	0	CLK/256
1	1	1	CLK/1024

Modo CTC

$$f_{OC2} = \frac{f_{osc}}{2N(1 + OCR2)}$$

Modo PWM rápido

$$f_{OC2_PWM} = \frac{f_{osc}}{N(1 + TOP)}$$

Modo PWM com fase corrigida

$$f_{OC2_PWM} = \frac{f_{osc}}{2N(1 + TOP)}$$

Configuração do modo de operação do TC2.

Modo	WGM22	WGM21	WGM20	Modo de Operação	TOP	Atualização de OCR2 no valor:	Sinalização do bit TOV2 no valor:
0	0	0	0	Normal	0xFF	Imediata	0xFF
1	0	0	1	PWM com fase corrigida	0xFF	0xFF	0x00
2	0	1	0	CTC	OCR2A	Imediata	OCR2A
3	0	1	1	PWM rápido	0xFF	0x00	0xFF
4	1	0	0	reservado	-	-	-
5	1	0	1	PWM com fase corrigida	OCR2A	OCR2A anterior	0x00
6	1	1	0	reservado	-	-	-
7	1	1	1	PWM rápido	OCR2A	0x00	OCR2A

Ação nas saídas OC2A/OC2B.

MODO NÃO PWM			MODO PWM RÁPIDO			MODO PWM FASE CORRIGIDA		
COM2A1 COM2B1	COM2A0 COM2B0	DESCRIÇÃO	COM2A1 COM2B1	COM2A0 COM2B0	DESCRIÇÃO	COM2A1 COM2B1	COM2A0 COM2B0	DESCRIÇÃO
0	0	OC2A/OC2B desconectado	0	0	OC2A/OC2B desconectado	0	0	OC2A/OC2B desconectado
0	1	Inverte OC2A/OC2B na comparação	0	1	Inverte OC2A na comparação*	0	1	Inverte OC2A na comparação*
1	0	OC2A/OC2B em zero na comparação	1	0	OC2A/OC2B = 0 na comparação, em 1 no final do período	1	0	OC2A/OC2B = 0 na comparação na contagem crescente, OC2A/OC2B = 1 na comparação na contagem decrescente.
1	1	OC2A/OC2B em alto na comparação	1	1	OC2A/OC2B = 1 na comparação, em 0 no final do período	1	1	OC2A/OC2B = 1 na comparação na contagem crescente, OC2A/OC2B=0 na comparação na contagem decrescente.

* Válido para WGM02 = 1, se WGM02 = 0: OC2A desconectado. COM0B1:0 = 1 é reservado.

TC1

TIMSK1 - Timer/Counter 1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
TIMSK1	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1
Lê/Escrive	L	L	L/E	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Bit = 1 – interrupção habilitada; Bit = 0, interrupção desabilitada.

ICIE1 – interrupção por entrada de captura

TOIE1 – interrupção por estouro de contagem.

OCIE1A, OCIE1B – interrupções por coincidência de comparação.

TIFR1 – Timer/Counter 1 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
TIFR1	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1
Lê/Escrive	L	L	L/E	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

Os bits sinalizam a solicitação de uma interrupção, são limpos pela escrita de 1 lógico na sua posição.

TCCR1A - Timer/Counter 1 Control Register A

Bit	7	6	5	4	3	2	1	0
TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10
Lê/Escr.	L/E	L/E	L/E	L/E	L	L	L/E	L/E
Valor Inic.	0	0	0	0	0	0	0	0

TCCR1B – Timer/Counter 1 Control Register B

Bit	7	6	5	4	3	2	1	0
TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
Lê/Escrive	L/E	L/E	L	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

WGM13:0 – configuração do modo de operação do TC1.

COM1A1:0 e COM1B1:0 – configuram a ação nas saídas OC1A/OC1B de acordo com o modo escolhido.

CS12:0 – seleção do *prescaler*.

ICNC1 = 1, habilita o filtro de ruído do pino de captura ICP1.

ICES1 = 1, evento de captura disparado por transição de 0 para 1 na entrada de captura; ICES1 = 0, pela transição de 1 para 0.

TCCR1C - Timer/Counter 1 Control Register C

Bit	7	6	5	4	3	2	1	0
TCCR1C	FOC1A	FOC1B	-	-	-	-	-	-
Lê/Escrive	L/E	L/E	L	L	L	L	L	L
Valor Inicial	0	0	0	0	0	0	0	0

FOC1A:B = 1, forçam uma comparação no módulo gerador de onda. As saídas são alteradas de acordo com os bits COM1A1:0 e COM1B1:0. Em zero nos modos PWM.

TCNTH e TCNTL (TCNT1) – Timer/Counter 1 Register

Registradores de 16 bits onde é realizada a contagem do TC1, H (*High*) e L (*Low*).

OCR1AH e OCR1AL (OCR1A) – Output Compare 1 Register A

Registradores de comparação A de 16 bits cada, H (*high*) e L (*low*).

OCR1BH e OCR1BL (OCR1B) – Output Compare 1 Register B

Registradores de comparação B de 8 bits cada, H (*high*) e L (*low*).

ICR1H e ICR1L (ICR1) – Input Capture Register 1

Esses registradores são atualizados com o valor do TCNT1 cada vez que um evento programado ocorre no pino ICP1 (ou opcionalmente nos pinos do comparador analógico). Também são empregados para definir o valor máximo de contagem (TOP).

Seleção do *clock* para o TC1

Frequência em OC1A/OC1B [Hz]

CS12	CS11	CS10	Descrição
0	0	0	TC1 parado
0	0	1	CLK
0	1	0	CLK/8
0	1	1	CLK/64
1	0	0	CLK/256
1	0	1	CLK/1024
1	1	0	Clock externo no pino T1. Borda de descida.
1	1	1	Clock externo no pino T1. Borda de subida.

Modo CTC

$$f_{OC1A} = \frac{f_{osc}}{2N(1 + TOP)}$$

Modo PWM rápido

$$f_{OC1x_PWM} = \frac{f_{osc}}{N(1 + TOP)}$$

Modo PWM com fase corrigida

$$f_{OC1A_PWM} = \frac{f_{osc}}{2N \cdot TOP}$$

Configuração dos modos de operação do TC1.

Modo	WGM13	WGM12	WGM11	WGM10	Modo de operação do TC1	Valor TOP	Atualiz. OCR1x no valor	Bit TOV1 ativo no valor:
0	0	0	0	0	Normal	0xFFFF	Imediata	0xFFFF
1	0	0	0	1	PWM c/ fase corrigida, 8 bits	0x00FF	0x00FF	0
2	0	0	1	0	PWM c/ fase corrigida, 9 bits	0x01FF	0x01FF	0
3	0	0	1	1	PWM c/ fase corrigida, 10 bits	0x03FF	0x03FF	0
4	0	1	0	0	CTC	OCR1A	Imediata	0xFFFF
5	0	1	0	1	PWM rápido, 8 bits	0x00FF	0	0x00FF
6	0	1	1	0	PWM rápido, 9 bits	0x01FF	0	0x01FF
7	0	1	1	1	PWM rápido, 10 bits	0x03FF	0	0x03FF
8	1	0	0	0	PWM c/ fase e freq. corrigidas	ICR1	0	0
9	1	0	0	1	PWM c/ fase e freq. corrigidas	OCR1A	0	0
10	1	0	1	0	PWM c/ fase corrigida	ICR1	ICR1	0
11	1	0	1	1	PWM c/ fase corrigida	OCR1A	OCR1A	0
12	1	1	0	0	CTC	ICR1	Imediata	0xFFFF
13	1	1	0	1	Reservado	-	-	-
14	1	1	1	0	PWM rápido	ICR1	0	ICR1
15	1	1	1	1	PWM rápido	OCR1A	0	OCR1A

Ação nas saídas OC1A/OC1B.

MODO NÃO PWM			MODO PWM RÁPIDO			MODO PWM FASE CORRIGIDA		
COM1A1 COM1B1	COM1A0 COM1B0	DESCRIÇÃO	COM1A1 COM1B1	COM1A0 COM1B0	DESCRIÇÃO	COM1A1 COM1B1	COM1A0 COM1B0	DESCRIÇÃO
0	0	OC1A/OC1B desconectado	0	0	OC1A/OC1B desconectado	0	0	OC1A/OC1B desconectado
0	1	Inverte OC1A/OC1B na comparação	0	1	Inverte OC1A na comparação nos modos 14 ou 15, OC1B desconectado*	0	1	Inverte OC1A na comparação nos modos 9 ou 11, OC1B desconectado*
1	0	OC1A/OC1B em zero na comparação	1	0	OC1A/OC1B = 0 na comparação, em 1 no final do período	1	0	OC1A/OC1B = 0 na comparação na contagem crescente, OC1A/OC1B = 1 na comparação na contagem decrescente.
1	1	OC1A/OC1B em alto na comparação	1	1	OC1A/OC1B = 1 na comparação, em 0 no final do período	1	1	OC1A/OC1B = 1 na comparação na contagem crescente, OC1A/OC1B=0 na comparação na contagem decrescente.

* Nos demais modos OC1A e OC1B desconectados.

SPI

SPCR – SPI Control Register

Bit	7	6	5	4	3	2	1	0
SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
Lê/Escr.	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

SPIE = 1, interrupção habilitada; **SPE** = 1, SPI habilitada; **DORD** = 1, transmite primeiro o LSB.

MSTR = 1, SPI como mestre; **SPR1** e **SPR0**, determinam a frequência de SCK.

Quando **CPOL** = 1 (polaridade do *clock*), SCK é alto quando inativo; quando CPOL = 0, SCK é baixo quando inativo.

O bit **CPHA** (fase do *clock*) determina se o dado é amostrado na borda principal ou na decida do sinal SCK.

SPSR – SPI Status Register

Bit	7	6	5	4	3	2	1	0
SPSR	SPIF	WCOL	-	-	-	-	-	SPI2X
Lê/Escrve	L	L	L	L	L	L	L	L/E
Valor Inicial	0	0	0	0	0	0	0	0

SPIF = 1, Transferência Completa; **WCOL** = 1, SPDR foi escrito durante uma transmissão.
SPI2X = 1, frequência de SCK duplicada.

SPDR – SPI Data Register

Bit	7	6	5	4	3	2	1	0
SPDR	MSB							LSB
Lê/Escr.	L/E							
Valor Inicial	X	X	X	X	X	X	X	X

Formato da transferência de dados.

Configuração		Borda Principal	Borda de Fuga	MODO SPI
CPOL	CPHA			
0	0	Amostragem (subida)	Ajuste (descida)	0
0	1	Ajuste (subida)	Amostragem (descida)	1
1	0	Amostragem (descida)	Ajuste (subida)	2
1	1	Ajuste (descida)	Amostragem (subida)	3

Frequência de operação para o modo mestre.

SPI2X	SPR1	SPR0	Frequência do SCK
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

f_{osc} = frequência de operação da CPU.

USART

UDR0 – USART I/O Data Register 0

Bit	7	6	5	4	3	2	1	0
UDR0 (leitura)	RXB[7:0]							
UDR0 (escrita)	TXB[7:0]							
Lê/Escr.	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

UCSR0A – USART Control and Status Register 0 A

Bit	7	6	5	4	3	2	1	0
UCSR0A	RXC0	TXC0	UDRE0	FE	DOR0	UPE0	U2X0	MPCM0
Lê/Escr.	L	L/E	L	L	L	L	L/E	L/E
Valor Inicial	0	0	1	0	0	0	0	0

RXC0 = 1, recepção completa, limpo quando UDR é lido.

TXC0 = 1, frame enviado, nenhum frame novo, limpo também pela escrita de 1.

UDRE0 = 1, UDR vazio, pronto para novo dado.

FE0 = 1, stop bit recebido = 0.

DOR0 = 1, UDR não foi lido e novo start bit é detectado.

P0E = 1, erro de paridade no byte recebido.

U2X0 = 1, dobra a taxa no modo assíncrono.

MPCM0 = 1, modo multiprocessador ativo.

UCSR0B – USART Control and Status Register 0 B

Bit	7	6	5	4	3	2	1	0
UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
Lê/Escr.	L/E	L/E	L/E	L/E	L/E	L/E	L	L/E
Valor Inicial	0	0	0	0	0	0	0	0

RXCIE0 = 1, habilita a interrupção por recepção completa.

TXCIE0 = 1, habilita a interrupção por transmissão completa.

UDRIE0 = 1, habilita a interrupção por UDR vazio.

RXEN0 = 1, habilita a recepção da USART.

TXEN0 = 1, habilita a transmissão da USART.

RXB80/TXB80 = nono bit recebido/transmitido.

UCSR0C - USART Control and Status Register 0 C

Bit	7	6	5	4	3	2	1	0
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOLO
Lê/Escr.	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	1	1	0

UMSEL01:0 - selecionam o modo de operação da USART.

UPM01:0 - selecionam o modo de paridade.

USBS0 = 1, dois bits de parada.

UCSZ01:0 - determinam o número de bits de um *frame*.

UCPOLO – válido para o modo síncrono, ajusta a relação entre a alteração do dado transmitido, a amostragem do dado recebido e o *clock* síncrono.

Ajuste do modo de operação da USART.

UMSEL01	UMSEL00	Modo de operação
0	0	assíncrono
0	1	síncrono
1	0	reservado
1	1	SPI mestre

Ajuste do modo de paridade.

UPM01	UPM00	Modo de Paridade
0	0	desabilitado
0	1	reservado
1	0	habilitado, paridade par
1	1	habilitado, paridade ímpar

Ajuste do Frame.

UCSZ02	UCSZ01	UCSZ00	Tamanho do Caractere
0	0	0	5 bits
0	0	1	6 bits
0	1	0	7 bits
0	1	1	8 bits
1	0	0	reservado
1	0	1	reservado
1	1	0	reservado
1	1	1	9 bits

Ajuste da polaridade do *clock* síncrono.

UCPOL0	Mudança do Dado Transmitido (saída do pino TxD0)	Amostragem do Dado Recebido (entrada do pino RxD0)
0	Borda de subida de XCK.	Borda de descida de XCK.
1	Borda de descida de XCK.	Borda de subida de XCK.

UBRR0L e UBRR0H – USART *Baud Rate Registers*

Bit	15	14	13	12	11	10	9	8				
UBRR0H	-	-	-	-	UBRR[11:8]							
UBRR0L	UBRR[7:0]											
Bit	7	6	5	4	3	2	1	0				
Lê/Escrive	L/E	L	L	L	L/E	L/E	L/E	L/E				
Valor Inicial	0	0	0	0	0	0	0	0				
	0	0	0	0	0	0	0	0				

Equações para o cálculo do registrador UBRR0 de taxa de transmissão.

Modo de operação	Equação para o cálculo da taxa de transmissão	Equação para o cálculo do valor de UBRR
Modo Normal Assíncrono (U2X0 = 0)	$TAXA = \frac{f_{osc}}{16(UBRR0 + 1)}$	$UBRR0 = \frac{f_{osc}}{16.TAXA} - 1$
Modo de Velocidade Dupla Assíncrono (U2X0 = 1)	$TAXA = \frac{f_{osc}}{8(UBRR0 + 1)}$	$UBRR0 = \frac{f_{osc}}{8.TAXA} - 1$
Modo Mestre Síncrono	$TAXA = \frac{f_{osc}}{2(UBRR0 + 1)}$	$UBRR0 = \frac{f_{osc}}{2.TAXA} - 1$

USART NO MODO SPI

UCSR0C - USART MSPIM Control and Status Register 0 C

Bit	7	6	5	4	3	2	1	0
UCSR0C	UMSEL01	UMSEL00	-	-	-	UDORD0	UCPHAO	UCPOL
Lê/Escr.	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	1	1	0

UMSEL01:0 - selecionam o modo de operação da USART.

UDORD0 = 1, transmite primeiro o LSB.

UCPHAO - determina se o dado é amostrado na borda principal ou na de descida do sinal SCK.

UCPOL0 = 1 , SCK é alto quando inativo; quando UCPOL0 = 0, SCK é baixo quando inativo.

Equações para calcular a taxa de transmissão da USART no modo SPI mestre.

Equação para o cálculo da taxa de transmissão	Equação para o cálculo do valor de UBRR0
$TAXA = \frac{f_{osc}}{2(UBRR0 + 1)}$	$UBRR0 = \frac{f_{osc}}{2 \cdot TAXA} - 1$

Formato da transferência de dados.

Configuração		Borda Principal	Borda de Fuga	MODO SPI
UCPOL0	UCPHAO			
0	0	Amostragem (subida)	Ajuste (descida)	0
0	1	Ajuste (subida)	Amostragem (descida)	1
1	0	Amostragem (descida)	Ajuste (subida)	2
1	1	Ajuste (descida)	Amostragem (subida)	3

TWI

TWBR – TWI Bit Rate Register

Bit	7	6	5	4	3	2	1	0
TWBR	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0
Lê/Escr.	L/E							
Valor Inicial	0	0	0	0	0	0	0	0

TWSR – TWI Status Register

Bit	7	6	5	4	3	2	1	0
TWBR	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0
Lê/Escr.	L/E							
Valor Inicial	0	0	0	0	0	0	0	0

TWSR7:3 - refletem o estado do TWI;

TWBR1:0 – determinam o *prescaler* do TWI.

Prescaler da taxa de bits do TWI.

Frequência do TWI em [Hz]

TWPS1	TWPS0	Valor do Prescaler
0	0	1
0	1	4
1	0	16
1	1	64

$$f_{SCL} = \frac{f_{osc}}{16 + (2 \times TWBR \times TWPS)}$$

TWCR – TWI Control Register

Bit	7	6	5	4	3	2	1	0
TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Lê/Escr.	L/E	L/E	L/E	L/E	L	L/E	L	L/E
Valor Inicial	0	0	0	0	0	0	0	0

TWINT – bit sinalizador de interrupção.

TWEA = 1, envia ACK; TWEA = 0, NACK.

TWSTA = 1, condição de início.

TWSTO = 1, condição de parada.

TWWC = 1, tentativa de escrita em TWDR com TWINT = 0.

TWEN = 1, interface ativa.

TWIE = 1, interrupção habilitada.

TWDR – TWI Data Register

Bit	7	6	5	4	3	2	1	0
TWDR	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0
Lê/Escr.	L/E							
Valor Inicial	1	1	1	1	1	1	1	1

TWAR – TWI (Slave) Address Register

Bit	7	6	5	4	3	2	1	0
TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
Lê/Escr.	L/E							
Valor Inicial	1	1	1	1	1	1	1	0

TWA6:0 – constituem o endereço da unidade TWI no modo escravo.

TWGCE = 1, habilita o reconhecimento de uma chamada geral no barramento TWI.

COMPARADOR ANALÓGICO

ADCSR B – ADC Control and Status Register B

Bit	7	6	5	4	3	2	1	0
ADCSR B	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0
Lê/Escr.	L	L/E	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

ACME = 1, com o AD desligado o multiplexador do AD seleciona a entrada (-) do comparador. Com ACME = 0, AIN1 é aplicado à entrada (-).

ACSR – Analog Comparator Control and Status Register

Bit	7	6	5	4	3	2	1	0
ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
Lê/Escr.	L/E	L/E	L	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	N/A	0	0	0	0	0

ACD = 1, comparador desabilitado.

ACBG = 1, Vref aplicada à entrada (+); ACBG = 0, AIN0 em (+).

ACO – saída do comparador. **ACI** – bit sinalizador de interrupção. **ACIE** = 1, habilita interrupção.

ACIC = 1, captura habilitada.

ACIS1:0 - determinam qual evento dispara a interrupção do comparador analógico.

Ajuste dos bits ACIS1 e ACIS0.

ACIS1	ACIS0	Modo de interrupção
0	0	Subida ou descida do sinal de comparação.
0	1	Reservado.
1	0	Borda de descida do sinal de comparação.
1	1	Borda de subida do sinal de comparação.

DIDR1 – Digital Input Disable Register 1

Bit	7	6	5	4	3	2	1	0
DIDR1	-	-	-	-	-	-	AIN1D	AIN0D
Lê/Escr.	L	L	L	L	L	L	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

AIN1D/AIN0D = 1, uso do comparador analógico ; AIN1D/AIN0D = 0 pinos como I/O.

Multiplexação da entrada negativa do comparador analógico.

ACME	ADEN	MUX2..0	Entrada Negativa para o Comparador Analógico
0	x	xxx	AIN1
1	1	xxx	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6 (nos encapsulamentos TQFP e QFN/MLF)
1	0	111	ADC7 (nos encapsulamentos TQFP e QFN/MLF)

ADC

ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0
ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0
Lê/Escr.	L/E	L/E	L/E	L	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

ADLAR = 1, resultado alinhado à esquerda; **ADLAR** = 0, alinhado à direita.

Seleção da tensão de referência do ADC.

REFS1	REFS0	Seleção da Tensão de Referência
0	0	AREF, tensão interna V_{REF} desligada.
0	1	AVCC, com um capacitor externo de 100 nF entre o pino AREF e o GND.
1	0	Reservado.
1	1	1,1 V interno, com um capacitor externo de 100 nF entre o pino AREF e o GND.

Seleção do canal de entrada.

MUX3..0	Entrada
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	Sensor interno de temperatura
1001-1101	reservado
1110	1,1 V (tensão fixa para referência)
1111	0 V (GND)

ADCL/ADCH – ADC Data Register

Bit		15	14	13	12	11	10	9	8
ADLAR=0	ADCH	-	-	-	-	-	-	ADC9	ADC8
	ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
Lê/Escr.		7	6	5	4	3	2	1	0
Valor Inicial		L	L	L	L	L	L	L	L
		L	L	L	L	L	L	L	L
		0	0	0	0	0	0	0	0
		0	0	0	0	0	0	0	0
Bit		15	14	13	12	11	10	9	8
ADLAR=1	ADCH	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
	ADCL	ADC1	ADC0	-	-	-	-	-	-
Lê/Escr.		7	6	5	4	3	2	1	0
Valor Inicial		L	L	L	L	L	L	L	L
		L	L	L	L	L	L	L	L
		0	0	0	0	0	0	0	0
		0	0	0	0	0	0	0	0

ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0
ADSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Lê/Escr.	L/E	L/E	L/E	L	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

ADEN = 1, ADC habilitado.

ADSC = 1, inicia a conversão.

ADATE = 1, modo de auto disparo.

ADIF – bit sinalizador de conversão completa.

ADIE = 1, interrupção habilitada.

Seleção da divisão de *clock* para o ADC.

ADPS2	ADPS1	ADPS0	Fator de Divisão
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

ADCSR_B – ADC Control and Status Register B

Bit	7	6	5	4	3	2	1	0
ADCSR _B	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0
Lê/Escr.	L	L/E	L	L	L	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

ADTS2:0 - se ADATE = 1 (registraror ADCSRA), o valor expresso pela combinação desses bits seleciona a fonte para o disparo da conversão; se ADATE = 0, esses bits não têm efeito.

Configurações dos bits ADTS2:0.

ADTS2	ADTS1	ADTS0	Fonte de disparo
0	0	0	Conversão contínua
0	0	1	Comparador Analógico
0	1	0	Interrupção Externa 0
0	1	1	Igualdade de comparação A do TC0
1	0	0	Estouro de contagem do TC0
1	0	1	Igualdade de comparação B do TC1
1	1	0	Estouro de contagem do TC1
1	1	1	Evento de captura do TC1

DIDR₀ – Digital Input Disable Register 0

Bit	7	6	5	4	3	2	1	0
DIDR ₀	-	-	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D
Lê/Escr.	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial	0	0	0	0	0	0	0	0

ADC5D:0D = 1, o pino correspondente é entrada para o ADC; ADC5D:ADC0D = 0, pinos como I/O.

H. ACRÔNIMOS

ABS	<i>Anti-lock Braking System</i>
ACK	<i>Acknowledge</i>
ADC	<i>Analog-to-Digital Converter</i>
ALU	<i>Arithmetic Logic Unit</i>
ARM	<i>Advanced RISC Machine</i>
ASCII	<i>American Standard Code for Information Interchange</i>
ASK	<i>Amplitude Shift Keying</i>
AVR	Alf – Vegard – RISC
BCD	Binário Codificado em Decimal
BGA	<i>Ball Grid Array</i>
BRTOS	<i>Brazilian Real-Time Operating System</i>
CAN	<i>Controller-Area Network</i>
CGROM	<i>Character Generator ROM</i>
CI	Círcuito Integrado
CISC	<i>Complex Instructions Set Computer</i>
CPU	<i>Central Processing Unit</i>
CRC	<i>Cyclic Redundancy Check</i>
CTC	<i>Clear Timer on Compare</i>
DAC	<i>Digital-to-Analog Converter</i>
DDRAM	<i>Data Display RAM</i>
DFN	<i>Dual Flat No-Lead Plastic Package</i>
DIL	<i>Dual In Line</i>
DIP	<i>Dual In line Package</i>
DMA	<i>Direct Memory Access</i>
DMIPS	<i>Dhrystone Million Instructions Per Second</i>
DSP	<i>Digital Signal Processing</i>
EEPROM	<i>Electrical Erasable Programming Read Only Memory</i>
ELF	<i>Executable and Linkable Format</i>
GCA	<i>General Call Address</i>
GCC	<i>GNU Compiler Collection</i>
GFSK	<i>Gaussian Frequency Shift Keying</i>
GND	<i>Ground</i>
GPOS	<i>General Purpose Operational System</i>
GPS	<i>Global Positioning System</i>
I2C	<i>Inter-IC (Integrated Circuit)</i>

IDE	<i>Integrated Development Environment</i>
IGBT	<i>Insulated Gate Bipolar Transistor</i>
I/O	<i>Input/Output</i>
ISM	<i>Industrial, Scientific and Medical</i>
ISP	<i>In-System Programming</i>
JTAG	<i>Joint Test Action Group</i>
LCD	<i>Liquid Crystal Display</i>
LDR	<i>Light Dependent Resistor</i>
LED	<i>Light Emitting Diodes</i>
LSB	<i>Least Significant Bit</i>
MIPS	<i>Million Instructions Per Second</i>
MISO	<i>Master Input Slave Output</i>
MLF	<i>Micro Lead Frame package</i>
MMC	<i>Multi Media Card</i>
MOSI	<i>Master Output Slave Input</i>
MR	<i>Master Receive</i>
MSB	<i>Most Significant Bit</i>
MSOP	<i>Mini Small Outline Plastic Packages</i>
MT	<i>Master Transmit</i>
NACK	<i>Not ACK</i>
OOK	<i>On/Off Keying</i>
OTP	<i>One Time Programmable</i>
PC	<i>Program Counter</i>
PCI	<i>Placa de Circuito Impresso</i>
PDA	<i>Personal Digital Assistants</i>
PDIP	<i>Plastic Dual Inline Package</i>
PTH	<i>Plated Through-Hole</i>
PWM	<i>Pulse Width Modulation</i>
QFN	<i>Quad Flat No leads package</i>
RAM	<i>Random Access Memory</i>
RFCOMM	<i>Radio Frequency Communications</i>
RGB	<i>Red-Green-Blue</i>
RISC	<i>Reduced Instruction Set Computer</i>
ROM	<i>Read Only Memory</i>
RS	<i>Register Select</i>
RTC	<i>Real Time Clock</i>
RTc	<i>Real Time counter</i>
RTOS	<i>Real Time Operational Systems</i>
RTTTL	<i>Ring Tones Text Transfer Language</i>

SD	<i>Secure Digital memory card</i>
SDHC	<i>High-Capacity SD Card</i>
SH	<i>Sample and Hold</i>
SIL	<i>Single In Line</i>
SIMD	<i>Single Instruction Multiple Data</i>
SMD	<i>Surface Mount Device</i>
SO	<i>Sistema Operacional</i>
SoC	<i>System-on-a-Chip,</i>
SOIC	<i>Small-Outline Integrated Circuit</i>
SP	<i>Stack Pointer</i>
SPI	<i>Serial Peripheral Interface</i>
SR	<i>Slave Receive</i>
SRAM	<i>Static RAM</i>
SS	<i>Slave Select</i>
ST	<i>Slave Transmit</i>
TC	<i>Timer/Counter</i>
TCB	<i>Task Control Block</i>
TQFP	<i>Thin profile plastic Quad Flat Package</i>
TWI	<i>Two Wire serial Interface</i>
USART	<i>Universal Synchronous and Asynchronous serial Receiver and Transmitter</i>
USB	<i>Universal Serial Bus</i>
USI	<i>Universal Serial Interface</i>
VAC	<i>Voltage Alternate Current</i>
VCC	<i>Voltage Continuous Current</i>
WDT	<i>Watchdog Timer</i>

