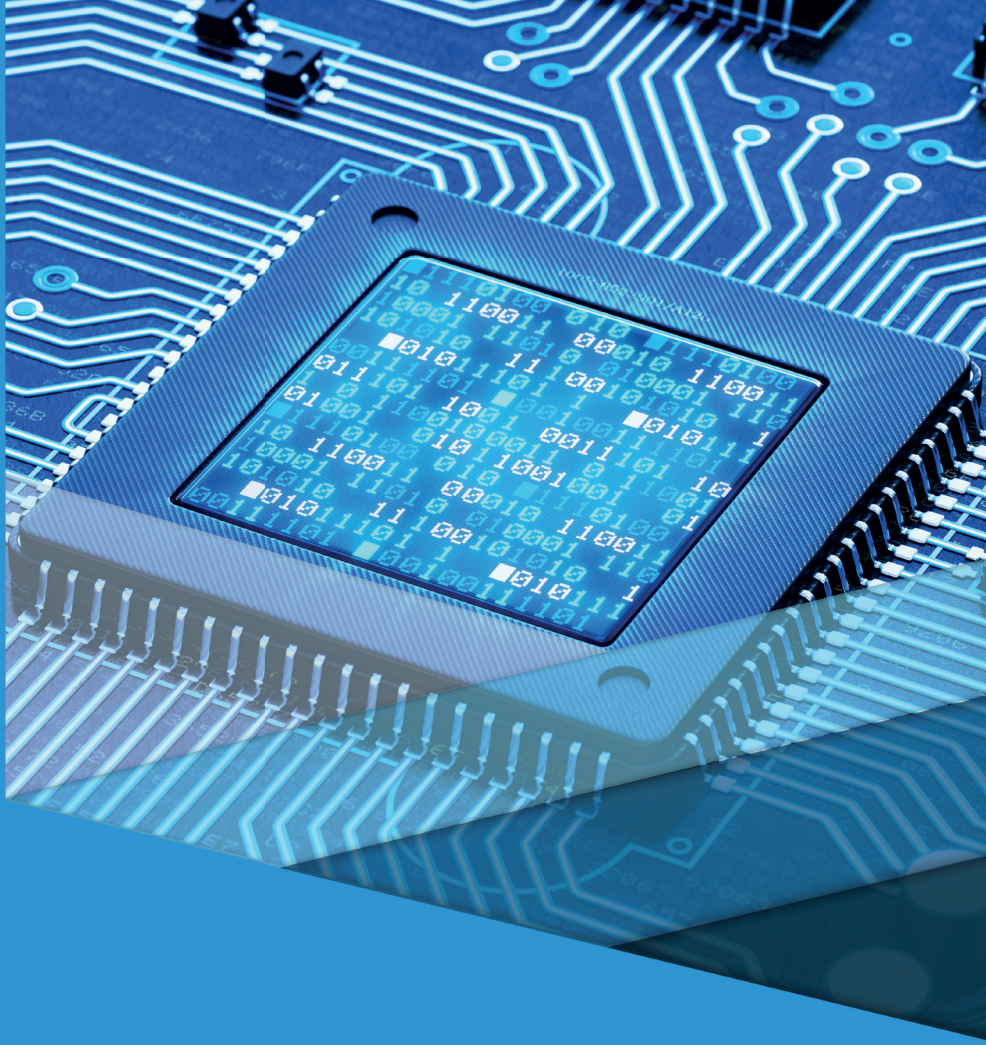


Microcontroladores e Microprocessadores

Microcontroladores e microprocessadores

Victor Gonçalves de Carvalho Feitosa Perim
Jefferson Nataline Rosa do Nascimento

Dados Internacionais de Catalogação na Publicação (CIP)

Perim, Victor Gonçalves de Carvalho Feitosa
P441m Microcontroladores e microprocessadores / Victor Gonçalves de Carvalho Feitosa Perim, Jefferson Nataline Rosa do Nascimento. – Londrina : Editora e Distribuidora Educacional S.A. 2017.
232 p.

ISBN 978-85-522-0273-8

1. Microcontroladores. 2. Microprocessadores – Programação. I. Nascimento, Jefferson Nataline Rosa. II.Titulo.

CDD 001.6404

2017
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: http://www.kroton.com.br/

Sumário

Palavras do autor

Caro aluno,

Todos nós já estamos acostumados com a utilidade de aparelhos eletrônicos no dia a dia. Até mesmo nossos avós, que conheceram um mundo totalmente sem eletrônicos e internet, inimaginável para os netos, estão conectados ao mundo virtual.

Pense em um mundo sem eletrônica. Difícil, não é mesmo? Pois saiba que a grande maioria desses produtos são desenvolvidos com o conhecimento que teremos nesta disciplina.

De maneira mais ampla, o conjunto de estudos sobre sistemas microprocessados foi dividido basicamente em duas partes: hardware e software, ambos estudados em vários cursos de Engenharia e Informática. A conexão dessas duas partes é fundamental na elaboração dos projetos aplicados para as mais diversas áreas e é estudada nesta disciplina de microcontroladores e microprocessadores, o que a torna fundamental para a formação dos profissionais que desejam atuar nessa área. Sendo assim, ao fim deste curso, cumprindo todas as tarefas, você estará pronto para desenvolver projetos de eletrônica avançados, com processamento de dados, controle de periféricos externos e comunicação entre dispositivos. Muitas áreas referentes à automação e eletrônica atuam a partir de microcontroladores e cada vez mais o profissional que domina bem essas áreas é mais requisitado no mercado de trabalho, principalmente na área de desenvolvimento de projetos de sistemas embarcados, considerando que estamos entrando na chamada Quarta Revolução Industrial, em que os sistemas ciberfísicos assumem autonomia para tomar decisões e se comunicar entre si e com os humanos.

Agora, você tem em mãos, de fato, um caminho para se tornar um excelente profissional, capaz de resolver vários problemas complexos desse ramo. Com esta finalidade, este livro foi escrito com muito empenho e dedicação e todos os detalhes foram pensados para que você, prezado aluno, possa aprender de maneira agradável e eficaz.

Este material está dividido em quatro unidades:

1. **Apresentação**: introdução dos principais conceitos sobre sistemas computacionais, sua história, aplicações, estrutura

interna básica e o funcionamento. Estudaremos em seguida os aspectos do chip escolhido, o ATmega328, bem como programá-lo em linguagem *Assembly*.

2. **Programação**: apresentação dos principais conceitos da linguagem C, voltada para programação de sistemas embarcados, e estudo dos fundamentos da linguagem, a estrutura de um programa, fluxogramas e funções e depois métodos de simulação.
3. **Periféricos básicos:** conhecimento dos periféricos mais básicos, presentes em praticamente qualquer projeto microprocessado, as portas digitais de entrada e saída, os temporizadores e as interrupções. Depois de entendidos, serão construídos projetos reais.
4. **Periféricos avançados**: estudo dos periféricos avançados mais importantes, como o conversor A/D, os módulos de comunicação e a memória EEPROM. Este conteúdo será colocado em prática em projetos mais avançados, com sinais analógicos e transmissão de dados.

Para completar nosso objetivo, precisamos de muito empenho no seu estudo!

Introdução aos sistemas embarcados

Convite ao estudo

Nesta primeira unidade, iniciaremos nossa jornada através do mundo digital/virtual, paralelo ao nosso, e recém-descoberto por nós. Vamos aprender como a eletrônica digital que você estudou pode ser organizada em memórias, processador, barramentos e periféricos, permitindo que um sistema processado exista e funcione. É muito importante um bom aproveitamento no estudo desta primeira parte, pois os fundamentos e todos os outros assuntos posteriores remeterão, direta ou indiretamente, aos termos apresentados aqui.

Depois de conhecer um pouco sobre a história do computador, sua evolução, seus periféricos internos e externos e suas principais aplicações e utilidades, estudaremos como ele funciona em sua essência, ou seja, conheceremos o seu interior. Para utilizar toda a teoria estudada em projetos práticos, empregados na indústria, foi escolhido o microcontrolador ATmega328, presente na popular placa Arduino UNO. Conheceremos quais são e como funcionam as suas partes internas, principalmente o núcleo e saberemos como criar programas em linguagem *Assembly*.

Esses conceitos abordados compõem todo o conteúdo teórico necessário para você adquirir as competências técnicas previstas para esta unidade, além de capacitá-lo para resolver as situações-problema enfrentadas. Novamente, pensando em tornar esse estudo mais próximo da realidade da indústria, embasaremos o conteúdo no seguinte cenário: uma montadora de veículos está no processo de criação de um novo automóvel e contratou algumas empresas para desenvolver as diferentes partes que compõem o veículo, como: motor, sistema de

transmissão mecânica, chassi, entre outros. Os módulos eletrônicos internos, que controlam o freio ABS, a injeção eletrônica e até o computador de bordo, serão construídos a partir de sistemas microprocessados e devem ser projetados individualmente, compondo a placa com os periféricos externos necessários.

Ao final desta unidade, você será capaz de avaliar os componentes destes módulos eletrônicos, bem como entender e explicar como os programas embarcados são executados e quais são as relações com as memórias e os periféricos internos e externos.

Considere agora que você é o engenheiro contratado para ser o responsável pela parte de eletrônica embarcada do projeto desta nova linha e deve estar preparado para tomar muitas decisões importantes, com sabedoria.

Seja muito bem-vindo! Estamos certos de que este será um ótimo estudo, esclarecedor e estimulante e esperamos que ao fim do curso você esteja ansioso para colocar em prática várias ideias que surgirão, como automatizar seu quarto ou sua casa.

Seção 1.1

Conceitos iniciais de microprocessamento

Diálogo aberto

Como já vivenciamos uma era de total conectividade, você provavelmente deve estar familiarizado com aparelhos eletrônicos, no ponto de vista de usuário, e o propósito aqui é que você comece a conhecer os sistemas como um engenheiro desenvolvedor. Inicialmente, aproveitando alguns conhecimentos já adquiridos no curso, estudaremos e entenderemos como é o funcionamento básico de um sistema computacional, bem como sua composição. Para consolidar esse primeiro aprendizado, consideraremos a situação proposta na apresentação da unidade, na qual você colocará em prática os conceitos estudados. Nessa situação, uma empresa montadora de veículos contratou você para projetar os módulos eletrônicos internos de um novo modelo. Sabemos que, apesar de parte destes módulos eletrônicos atuarem a partir de um programa, todos os carros possuem alguns módulos ausentes de processamento. Isso também ocorre em nossas residências, como o ventilador, as torradeiras e os liquidificadores mais simples não são considerados sistemas embarcados, ao contrário dos televisores digitais, máquinas de lavar e secretárias eletrônicas. Então, por que alguns aparelhos elétricos são projetados para atuar sob um programa computacional, e outros não? No caso do carro, quais critérios você usará para definir quais serão os módulos microprocessados? Quais aspectos do projeto devem ser ponderados para optar por um sistema microprocessado? Para um sistema embarcado, quais são as considerações para escolher entre um microprocessador, um microcontrolador ou até mesmo um hardware digital (FPGA – *Field Programmable Gate Array*, em português Arranjo de Portas Programáveis em Campo)?

Não pode faltar

Ao atingir o pensamento lógico, racional e abstrato, o homem demonstrou necessidade de simbolizar, quantificar e classificar as coisas ao seu redor, para realizar suas tarefas do dia a dia, cada vez

mais elaboradas. Utilizamos o sistema de números há muitos anos, antes mesmo de eles ganharem nome, como quando o pastor usava uma pequena corda com nós para saber quantas ovelhas tinha. Depois da formalização da Matemática e modelagem de fenômenos físicos, surgiu a ideia de automatizar um trabalho ou um processo de operação matemática, como a soma de dois números naturais, mas a partir de elementos da natureza, devidamente combinados, ou seja, uma máquina ao invés de um ser humano.

As primeiras máquinas para realizar processamentos, como a calculadora, ainda não tinham o nome de computador e foram inicialmente mecânicas e até hidráulicas, antes de se tornarem eletrônicas. Inclusive computadores analógicos foram desenvolvidos em paralelo com os digitais, ou discretos. O nome *computers* era usado, no início do século XX, para se remeter a funcionários, na grande maioria mulheres, que realizavam tarefas como calcular e processar informações, ou seja, computar (GRIER, 2005).

A computação moderna é resultado de várias transformações, sendo moldada pelo avanço da ciência e da tecnologia. Muitas descobertas contribuíram para isso, mesmo sem a perspectiva de quanto estas ajudariam no avanço das máquinas. A lógica matemática, por exemplo, foi consolidada por Gottfriend W. Leibniz, em 1702, e serviu para *George Boole* formular a álgebra booleana, usada nas operações computacionais hoje (IFRAH, 2001). No entanto, antes de o computador assumir essa forma que conhecemos, muitas propostas diferentes foram apresentadas, sendo que algumas realmente cumpriram seu papel esperado e serviram de inspiração para futuras gerações. As calculadoras mecânicas, como a de Pascal, *Pascaline*, aceleravam o trabalho dos calculistas, mas ainda se mostravam lentas e limitadas. A era da computação digital iniciou sua forte ascendência depois da Segunda Guerra Mundial, com os computadores eletromecânicos, os quais usavam relés mecânicos para realizar seus cálculos e tiveram como protagonista o sistema *Mark I*.

Em 1943, cientistas americanos começaram o projeto do ENIAC – *Electronic Numerical Integrator and Computer*, ou computador integrador numérico eletrônico, o primeiro computador eletrônico digital fabricado em escala (REGAN, 2012). Ele era chamado de "cérebro gigante" e era capaz de resolver muitas classes de problemas numéricos através de reprogramação do seu sistema operacional,

armazenado em cartões perfurados. Comandado por uma máquina de Turin completa, realizava seus cálculos a partir de 7500 tubos de vácuo e 17000 válvulas termiônicas e pesava 27 toneladas.

Essa primeira calculadora eletrônica usava aritmética decimal ao invés de binária, servindo inicialmente no processamento de equações diferenciais complexas para calcular trajetórias balísticas para o exército. Anteriormente, seu trabalho era feito por um humano, que demorava cerca de 2400 vezes mais tempo que o ENIAC. Esse projeto deu origem ao EDVAC – *Electronic Discrete Variable Automatic Computer*, ou computador eletrônico automático de variável discreta, que utilizava sistema de lógica binária. Foi construído pelos mesmos criadores do projeto ENIAC, visando aperfeiçoar vários problemas encontrados, além de receber a contribuição de *John von Neumann*, um matemático húngaro de origem judaica, naturalizado estadunidense. Esta foi a primeira vez que o cientista implementou sua arquitetura: ***Von Neumann***, que se tornou um padrão para os futuros computadores durante muitos anos, até surgir a arquitetura ***Harvard***, que será estudada nesta seção.

Esse sistema contribuiu muito para a disseminação da computação, pois já era usado, além das forças militares, pelo governo e algumas indústrias para realizar grandes processamentos. No entanto, o uso do computador se tornou realmente livre quando, em 1951, foi apresentado o UNIVAC – *Universal Automatic Computer*, ou computador automático universal, o primeiro computador comercial produzido, desenvolvido novamente pelos mesmos cientistas do ENIAC (CERUZZI, 2003). Também operava a partir de programas gravados em cartões perfurados e passou a receber as mais diversas utilidades de processamento, apesar de ocupar uma sala de mais de 35 m^2. Depois da invenção e consolidação dos transistores, estes substituíram os tubos de vácuo nos projetos de computação e as memórias de armazenamento magnético ou elétrico deram fim ao uso de cartões perfurados para armazenar informação, gerando a segunda geração de computadores. Percebendo o suposto enorme avanço que isso traria, iniciou-se no mundo todo uma corrida científica em busca de miniaturização e otimização em velocidade e energia para sistemas digitais com semicondutores, ou seja, transistores e memórias. Dessa forma, começaram a surgir os primeiros circuitos integrados, os chips, que podiam realizar processamentos avançados, consumindo pouca energia, tempo e espaço e que se

tornaram cada vez mais baratos, com a produção em larga escala. Essa busca por incrementar e otimizar o computador parece não ter fim, uma vez que, apesar de já dominarmos a computação para o auxílio em muitas necessidades básicas, ainda existem novas ideias e aplicações aguardando para se consolidar, como uma maior velocidade de processamento ou o computador quântico. Estes circuitos computacionais integrados também ganharam circuitos adicionais específicos, reunindo todo o hardware necessário para a construção de um projeto, em uma única placa ou chip. Com a flexibilidade do programador escolher o comportamento do software e o custo/tempo de projetos serem cada vez menores, os sistemas computacionais foram absorvidos pela humanidade, o que vivenciamos a partir das últimas décadas.

Computador, microprocessador e microcontrolador

Os sistemas embarcados, apesar de serem uma tecnologia moderna, já estão disseminados e são amplamente utilizados em nosso cotidiano. Praticamente todos os aparelhos eletrônicos que compramos hoje são microprocessados ou controlados digitalmente. Esses aparelhos podem ser controlados por três núcleos distintos: um microprocessador, um microcontrolador ou um sistema digital sem software (FPGA). Já os dispositivos digitais mais complexos, denominados computadores, são basicamente todos feitos com microprocessadores com os periféricos.

Assimile

Um sistema embarcado é um dispositivo eletrônico e computacional capaz de controlar periféricos a partir da execução de um programa, que não seja um computador. É projetado para realizar tarefas bem particulares, com grande restrição de recursos, memória e processamento.

O computador é geralmente muito mais caro que um sistema embarcado, tem finalidades variadas e grande capacidade de processamento e armazenamento. As calculadoras, por exemplo, foram projetadas a partir de um processador para apenas calcular e exibir os valores e com o mínimo de recursos possível para isto. Já um computador, além de também calcular, pode realizar

milhares de outras tarefas. Desta forma, apesar de ser um sistema computacional, classificamos uma calculadora como um sistema embarcado e não um computador. Por outro lado, um sistema eletrônico ausente de processamento, como um amplificador de som, não é considerado um sistema embarcado. No entanto, por que alguns aparelhos possuem microcontroladores e outros não? Na verdade, a definição do produto é baseada nas necessidades de processamento de dados e controle avançado. Um liquidificador, por exemplo, devido à natureza de sua função, não necessita de nenhum controle eletrônico, pois o chaveamento mecânico dos botões já é suficiente para encontrar a velocidade adequada. Já o micro-ondas, pela responsabilidade de ler as teclas, controlar o tempo e a potência de acionamento, é um sistema embarcado, uma vez que o mesmo sistema sem processamento teria muito mais componentes e seria, consequentemente, mais caro e pior de ser construído.

Exemplificando

Se pararmos para observar, podemos facilmente identificar quais dos aparelhos que utilizamos possuem processamento, como o televisor digital, os celulares e câmeras digitais, secretária eletrônica, micro-ondas, MP3 player, carros, impressoras, máquina de lavar, entre muitos outros. No entanto, alguns aparelhos não necessitam de processamento, como: fogão, liquidificador, amplificador de som, interruptor, chuveiro, escova de dentes eletrônica etc.

Os microprocessadores são mais antigos que os microcontroladores e, de fato, é difícil falar em diferença entre eles, pois dentro de todo microcontrolador há um microprocessador, além de outros componentes, e eles operam em aplicações muito semelhantes. O conceito sobre a diferença é basicamente: o microprocessador não possui periféricos e memória interna, ao contrário do microcontrolador. Os microcontroladores são na verdade uma evolução dos microprocessadores, uma vez que se originaram devido à demanda da criação de SOCs – *System On Chip* (sistemas completos em um único chip) para acelerar o desenvolvimento de soluções para controle eletrônico na indústria. O processador não é capaz de armazenar dados, acionar periféricos, nem mesmo de tomar decisões sozinho, sem um

programa para conduzi-lo. Seus componentes internos são os registradores de trabalho, unidade de controle central, a unidade lógica e aritmética, ponteiros e registrador de controle e status. Para obter um sistema capaz de executar um programa para o controle de algum periférico, o processador deve se unir a outros elementos através de barramentos, operando em conjunto. O código fica em uma memória não volátil e controla o UCP – Unidade Central de Processamento (o microprocessador). Por meio deste, o programa comanda também as memórias e periféricos internos, realizando as trocas de dados e controles da maneira correta, como foi elaborado pelo programador do código. O seu computador pessoal é um exemplo disso, pois apesar de ser um só, tem suas partes internas separadas, conectadas por barramentos externos. Você com certeza já ouviu falar da memória RAM, do HD, da placa de vídeo ou do processador. São os componentes que, juntos, fazem seu computador funcionar plenamente, e que muitas vezes são considerados periféricos internos, por estarem todos dentro do gabinete. Para nós, corretamente, esses componentes são classificados como **periféricos externos**, pois não foram construídos dentro de um único chip.

Reflita

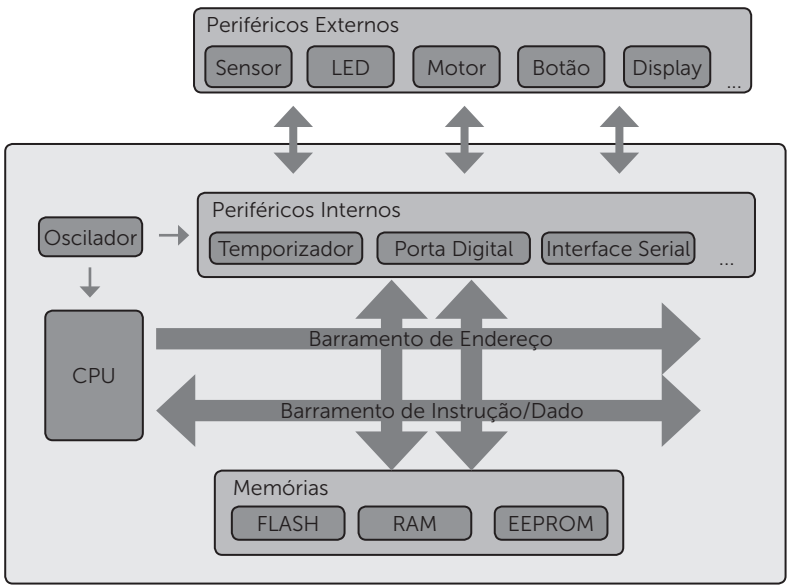
Denominamos **periféricos internos** os circuitos que são embutidos dentro do chip com o processador e se comunicam por barramentos internos.

Pense: quais são as razões para se produzir computadores miniaturizados em um único circuito integrado? Quais são as restrições desses sistemas, comparados aos computadores pessoais?

Com a evolução da tecnologia, a ciência e a indústria trabalharam na integração desses circuitos dedicados em um único chip, gerando o microcontrolador, capaz de controlar periféricos externos de forma autônoma. Os microcontroladores são perfeitos para a construção de sistemas automáticos, pois são muito mais baratos que um computador (geralmente menos que U$1,00), tem todos os recursos necessários para funcionar embutidos e são fáceis de ser projetados. Atualmente há inúmeros modelos de microcontroladores, o que permite uma escolha apropriada para cada projeto, tendo em vista quais periféricos internos e quanto

de memórias será necessário. A forma com que esses periféricos internos estão dispostos e conectados é denominada **arquitetura computacional** e um modelo básico pode ser visto na Figura 1.1.

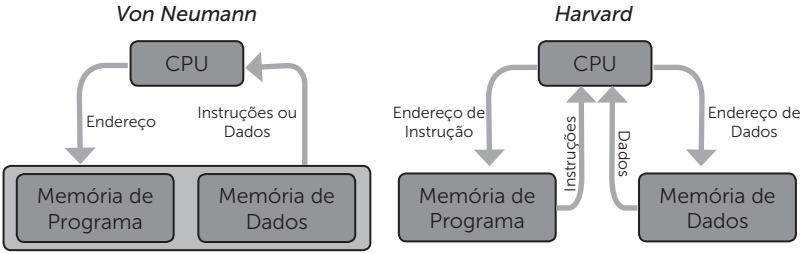
Figura 1.1 | Arquitetura básica de um microcontrolador



Fonte: elaborada pelo autor.

Apesar de muitas arquiteturas diferentes terem sido apresentadas e implementadas com sucesso, apenas duas delas se tornaram um padrão e ditaram os rumos da computação: a **Harvard** e a **Von Neumann**. Esta última foi revelada pelo físico e matemático húngaro-americano John von Neumann, que publicou seu modelo de arquitetura no documento *Primeiro rascunho de um relatório sobre o EDVAC*, em 30 de junho de 1945 (REGAN, 2012). Essa arquitetura é caracterizada pelo compartilhamento das instruções (ou programa) e dados em um único espaço de memória, além de utilizarem o mesmo barramento para transporte. Dessa forma, os elementos internos se comunicam através de dois barramentos: um para instruções e dados e outro para endereços, além dos outros dois essenciais, de controle e de relógio, ocultados na Figura 1.2.

Figura 1.2 | Arquiteturas de computador Von Neumann e Harvard



Fonte: elaborada pelo autor.

O fato das instruções e dados compartilharem a mesma memória e barramento tornou-se um fator limitante para aplicações que precisavam de respostas mais rápidas, pois a busca de dados e de instruções não podem ocorrer ao mesmo tempo. Pensando nisso, foi criada a arquitetura Harvard, um modelo que armazena instruções e dados em memórias diferentes, com barramentos individuais, permitindo que instruções e dados sejam recebidos paralelamente pelo núcleo. Depois este modelo, como o anterior, recebeu a capacidade de processar uma instrução enquanto a próxima está sendo lida, um processo denominado *pipeline*, ou canalização.

Pesquise mais

O *pipeline* é uma técnica de processamento que trouxe muitos benefícios para os processadores modernos. Pesquise mais sobre esse procedimento e entenda como é seu funcionamento básico. Disponível em: <http://homepages.dcc.ufmg.br/~brunors/AOCII/pipeline.pdf>. Acesso em: 8 jun. 2017.

Atualmente, a grande maioria dos processadores implementa a arquitetura de *Harvard* modificada, muito parecida com a primária, mas com adaptações para otimizar a performance. Esta arquitetura é encontrada nos microcontroladores ARM – Advanced Risc Machine (ou máquina avançada de conjunto de instruções reduzido) e processadores x86, bem como no ATmega328.

Memórias

Apesar de conter outros significados, para a computação, memória significa componente de hardware capaz de armazenar

informação digital, por tempo indeterminado, ou temporariamente, apenas enquanto estiver ligada. As memórias são classificadas como:

- EEPROM: atuais memórias de dados permanentes, que podem ser apagadas e escritas eletricamente. São derivadas das primeiras memórias, que só podiam ser gravadas no momento da fabricação e por isso se chamavam ROM – *Read Only Memory*, ou memória de apenas leitura. Posteriormente surgiu a PROM, adicionando a letra 'P' de *programmable*, ou programável, que só pode ser gravada uma única vez. Surgiu então a EPROM que pode ser gravada mais de uma vez, com apagamento por luz UV e, por fim, a memória *Electrically-Erasable Programmable Read-Only Memory*, ou memória somente para leitura eletricamente apagável e programável.

- RAM: são as memórias que armazenam dados temporariamente, ou seja, memória volátil. Significa *Random Access Memory*, ou memória de acesso aleatório. Permitem um acesso muito mais rápido, otimizando a resposta de processamento. Dessa forma, os dados que serão utilizados apenas um breve período e depois de processados perdem a utilidade, ficam armazenados na RAM, bem como os programas temporários. As primeiras memórias RAM não eram capazes de manter seu nível de tensão lógico alto por muito tempo e precisavam ter seus valores reatualizados. Atualmente, as memórias RAM são estáticas, chamadas de SRAM (*Static Random Access Memory*) ou memória estática de acesso aleatório. Também existem as DDRAM – *Dynamic Random Access Memory*, que são as memórias dinâmicas, presentes nos computadores e microcontroladores mais avançados.

- FLASH: este tipo de memória é um modelo híbrido dos outros dois citados, apesar de ser classificada como EEPROM e é não volátil. Foi desenvolvida pela empresa Toshiba na década de 1980 e possui um tempo de acesso menor que a antecessora EEPROM, embora não tão rápido como a RAM. É a memória de programa, onde fica armazenado o código.

Periféricos

Assim como os microprocessadores, os microcontroladores também precisam de periféricos para atuarem propriamente na solução de um problema, apesar de já possuir internamente alguns essenciais. Sendo assim, esses circuitos, amplamente utilizados com microprocessadores, foram integrados ao chip, gerando o microcontrolador, são chamados de **periféricos internos**. Estes são usados para "traduzir" os sinais exteriores ao chip para o seu núcleo digital e garantem o funcionamento central do sistema, mas, visto de uma maneira geral, não são suficientes para criar uma solução completa. O papel fundamental dos sistemas embarcados é controlar elementos da natureza (estado das saídas), a partir dos sinais de entrada, que também representam o estado do ambiente controlado. Esses valores devem ser transcritos do mundo físico para o mundo virtual, por meio de sinais elétricos e vice-versa e essa é a função dos **periféricos externos**.

Exemplificando

Até mesmo os projetos mais primários precisarão de periféricos externos. O sistema de um semáforo, por exemplo, utilizará minimamente os periféricos externos: botão de pedestre e luzes do farol, além dos periféricos internos: temporizadores e entradas/saídas digitais.

Atualmente os fabricantes oferecem diversos modelos de microcontroladores, agrupados em famílias, em que cada um possui um seleto grupo de periféricos internos, pensado para atender a um tipo de problema específico. Isso é muito útil na hora de escolher o modelo apropriado para cada projeto, pois certamente existe algum que contempla apenas os periféricos e tamanho de memória necessários para a aplicação, evitando desperdício de hardware e custo.

Reflita

Considerando o seu conhecimento prévio e o pouco que vimos até agora, você saberia descrever quais seriam os periféricos internos e externos necessários no projeto de um módulo de som automotivo, o rádio digital que temos nos carros atuais?

Os principais módulos ou periféricos internos encontrados nos microcontroladores são:

- **Circuito oscilador**: é a fonte de sinais de relógio, usados tanto pelo processador quanto pelos periféricos internos ou externos, que precisam de referência temporal ou sincronismo.
- **Memória RAM**: local onde são armazenadas as variáveis declaradas no programa.
- **Memória FLASH**: área em que será guardado o programa no formato de "linguagem de máquina". Também pode ser usada para guardar constantes.
- **Memória EEPROM**: é acessada quando se deseja armazenar dados por períodos indeterminados, como a senha de um usuário ou os coeficientes de um filtro.
- **Portas digitais**: são os módulos responsáveis por configurar e gerenciar os canais digitais de entrada e saída.
- **Temporizadores**: são os circuitos que fornecem ao programa meios de escolher o intervalo de tempo entre ações, permitindo o sincronismo temporal de tarefas concomitantes.
- **Módulo PWM**: do Inglês *Pulse Width Modulation*, ou pulso com modulação de largura, é um periférico que pode gerar sinais de PWM, muito úteis para o acionamento gradual de cargas.
- **Conversor analógico-digital**: capaz de fazer a aquisição e a digitalização de sinais analógicos por amostragem e comparações sucessivas, respectivamente, fornecendo os valores para o programa principal processar.
- **Conversor digital-analógico**: tem o mesmo papel do periférico anterior, mas de forma invertida.
- **Interrupções**: presente em alguns casos dentro do CPU, este módulo é responsável por configurar e gerenciar os sinais de interrupção.
- **Módulos de comunicação**: a grande maioria opera com transmissão serial, que pode ser síncrona ou assíncrona. Servem para transmitir e receber informações de sistemas externos, permitindo operações conjuntas e distribuídas. Os mais encontrados são: **UART** – *Universal Asynchronous Receiver/Transmitter*, ou transmissor/receptor universal

assíncrono; SPI – *Serial Peripheral Interface* (ou interface periférica serial) e I^2C - Inter-Integrated Circuit (ou circuito interintegrado).

Provavelmente, alguns desses periféricos internos são novidade para você, mas, quando observamos os periféricos externos, percebemos que estamos bem familiarizados com eles, pois já somos usuários dos aparelhos eletrônicos há um bom tempo.

Exemplificando

Pensando nos sistemas computacionais de uma maneira geral, ou seja, sistemas embarcados e computadores, os periféricos externos mais comuns são: botão, tecla e teclado, resistor, LED (*light-emitting diode*, ou diodo emissor de luz), display, mostradores, mouse, memória externa, conversor A/D ou D/A, *buffer* ou *driver* de corrente, optoacoplador, transistor, relé, contator, motor, sensor digital ou analógico, impressora, scanner, controle remoto, leitora de código de barras, leitor de impressão digital, *touchpad*, câmera digital, headphone, módulos sem fio (wifi, bluetooth, 4G etc.), MP3 player, pendrive, cartão de memória, caixa de som, entre muitos outros.

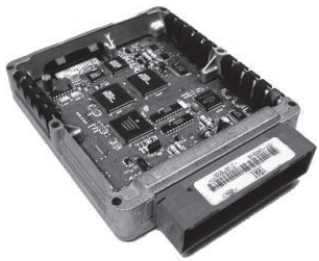
Sem medo de errar

Lembre-se de que você é o responsável por projetar os módulos eletrônicos internos de um novo modelo de veículo. Sabemos que, apesar de parte destes módulos eletrônicos atuarem a partir de um programa, todos os carros possuem alguns módulos ausentes de processamento. Por que alguns aparelhos elétricos são projetados para atuar sob um programa computacional e outros não? No caso do carro, quais critérios você usará para definir quais serão os módulos microprocessados? Quais aspectos do projeto devem ser considerados para optar por um sistema microprocessado? Para um sistema embarcado, quais são as considerações para escolher entre um microprocessador, um microcontrolador ou até mesmo um hardware digital (FPGA)?

Os módulos eletrônicos presentes no automóvel devem possuir, ou não, um microcontrolador de acordo com a necessidade de

processamento ou até mesmo de comunicação avançada com outros conjuntos.

Figura 1.3 | Módulo automotivo microprocessado para injeção eletrônica



Fonte: <https://goo.gl/GAqMtg>. Acesso em: 22 abr. 2017.

Os interruptores dos vidros elétricos, por exemplo, apesar de se comunicarem com os motores acionadores dos vidros, não necessitam de processamento, uma vez que a comunicação é primária, informando somente os estados liga, desliga e sentido, e isso pode ser feito através de apenas três fios e comutadores elétricos. No entanto, alguns desses módulos possuem a função de antiesmagamento que, através de um periférico externo, um sensor de corrente do motor do vidro consegue detectar um aumento na corrente do motor devido ao bloqueio do vidro através de um periférico interno, conversor A/D. Já os módulos de freio ABS, por exemplo, precisam de um controle mais apurado para o acionamento dos freios e de uma comunicação mais complexa para permitir o sincronismo eficaz dos quatro atuadores. Hoje em dia, praticamente todos os sistemas embarcados são feitos a partir de um microcontrolador, devido à sua praticidade, fácil implementação, robustez e custo.

Os microprocessadores são utilizados em sistemas com maior necessidade de processamento, maior complexidade, que se aproximam mais de um computador. Esses aparelhos são bem mais custosos e geralmente não são usados na automação de controle simples, mas geralmente em processamentos intensos de imagem, som ou grandes massas de dados, como um videogame moderno, de alta resolução gráfica.

Pensando agora no problema do carro, o único elemento que poderia usar um microprocessador seria o computador de bordo e caso seja detectado que apenas um microcontrolador não será o

suficiente para colher e processar todas as informações do veículo. Para um sistema embarcado criado a partir de hardware digital, ou seja, FPGAs, sua utilidade fica ainda mais restrita. Esses sistemas são geralmente protótipos, usados como base para construir chips posteriormente ou para casos que demandam velocidade de resposta superior ao de programas embarcados, ou paralelismo, como em alguns grandes roteadores. Você pode explorar mais o estudo sobre hardware digital em *Descrição e síntese de circuitos digitais* (D'AMORE, 2012).

Alguns aparelhos, apesar de aparentemente não apresentarem necessidade, incorporaram um microcontrolador. Isso se dá pela necessidade de inovação e sofisticação dos sistemas. Uma torradeira, por exemplo, pode ter sua temperatura controlada digitalmente, além de mostrar quantos segundos faltam para a torrada estar pronta. Um vendedor que faz essa escolha tenta agregar valor ao seu produto e destacá-lo entre os concorrentes, por oferecer mais recursos. Apesar da adição de um microcontrolador aumentar o custo de um produto, este pode ser vendido por muito mais devido aos benefícios adquiridos.

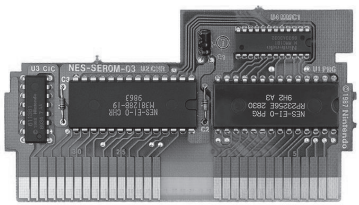
Avançando na prática

Definição do núcleo de um sistema embarcado

Descrição da situação-problema

Uma empresa de desenvolvimento de jogos contratou você para gerenciar o desenvolvimento de um novo modelo de videogame portátil de bolso, construído a partir de um sistema microprocessado: o famoso jogo Tetris. Para tal sistema, as entradas consideradas devem ser os sinais dos botões do controle, a chave liga e desliga e obviamente os sinais de relógio e de energia. As saídas a serem controladas são o display de LCD - *Liquid-Crystal Display* (ou mostrador de cristal líquido), que exibirá a imagem do jogo, dois LEDs que sinalizam ligado/desligado e jogo em *pause*, ou parado, além do sinal de áudio, que deve ser enviado para os alto-falantes do jogo ou para o conector de *headphone*.

Figura 1.4 | Exemplar de uma placa-cartão (núcleo apenas) para o jogo Tetris



Fonte: <https://goo.gl/TqnWYb>. Acesso em: 22 abr. 2017.

O seu primeiro passo é definir os componentes usados neste projeto, começando pelo mais importante: o núcleo. Sabendo que este não será um computador, pois não precisa de tantos recursos/custo, porém, deve ser um sistema embarcado, pois necessita de processamento, o projeto deve ser construído a partir de um microprocessador, um microcontrolador, uma FPGA, ou não faz diferença. Por quê? Quais são as considerações relevantes para essa decisão sempre feita na criação de um sistema embarcado?

Resolução da situação-problema

Podemos afirmar que existem diferenças em vários aspectos de tal escolha, como a quantidade de processamento ou o custo final. Pelo fato deste tipo de jogo não exigir gráficos elaborados e nem elevados níveis de processamento, pode-se afirmar que um microcontrolador seria suficiente para responder a todas as requisições em tempo real. A escolha de um microprocessador traria a necessidade de anexar todos os periféricos necessários para compor o conjunto mínimo, que no caso seriam os circuitos de: controle de energia e hibernação; oscilação e geração de sinais de relógio; temporização; buffer de interface digital (portas digitais); memória de programa; memória de dados volátil; memória de dados não volátil e de conexão entre todos eles (barramentos). Isso traria muita complexidade na construção do sistema, aumentando a demora, o custo e as chances de defeitos. O uso de uma FPGA tornaria o projeto também muito mais complexo e, consequentemente, caro de ser projetado. Entretanto, vale ressaltar que, se houver demanda de milhares de unidades, a criação de um hardware digital dedicado pode reduzir custos.

Faça valer a pena

1. Dentro de um microcontrolador, considerado um sistema completo ou autossuficiente, estão inclusos algumas memórias e periféricos de entrada/saída, os quais armazenam e transmitem informações constantemente.
Os programas executados no microcontrolador são armazenados:

a) Na memória RAM.
b) Na memória FLASH.
c) Nos barramentos.
d) Nos periféricos.
e) No CPU.

2. Apesar de algumas arquiteturas computacionais diferentes terem sido apresentadas e implementadas com sucesso, apenas duas delas se tornaram um padrão e ditaram os rumos da computação: a Harvard e a Von Neumann.
A arquitetura de computadores Harvard contornou o limitante do modelo de Von Neumann, permitindo uma geração de processadores muito mais poderosa. É correto afirmar que isto se deu na arquitetura Harvard devido:

a) A programas e instruções usarem espaços de memória e barramentos distintos, permitindo paralelismo na buscas destes.
b) Ao processador possuir mais de um núcleo, possibilitando processamento paralelo.
c) Ter sido implantado o uso da memória RAM, acelerando o acesso aos dados.
d) Às memórias volátil e não volátil serem unificadas em um único espaço de memória, simplificando o acesso.
e) Aos processadores trabalharem em rede, otimizando o processamento geral.

3. Com a evolução da tecnologia, a ciência e a indústria trabalharam na integração de alguns circuitos dedicados em um único chip, gerando o microcontrolador, capaz de controlar periféricos externos de forma autônoma.
Qual dos componentes pode ser considerado um periférico interno?

a) LED.
b) Botão.
c) Portas digitais.
d) Display.
e) Sensor fim-de-curso.

Seção 1.2

Arquitetura AVR – ATmega328

Diálogo aberto

Agora que já sabemos os conceitos básicos sobre sistemas computacionais, começaremos a aprofundar nosso estudo, conhecendo o núcleo e a estrutura interna do microcontrolador ATmega328 e o chip da placa Arduino UNO. O conhecimento destas partes internas é muito importante para os programadores de sistemas embarcados, para saber quais recursos estão disponíveis, como são configurados e como atuam no sistema. Por exemplo, é fundamental para um desenvolvedor que programa em linguagem *Assembly* saber que o espaço de memória RAM para variáveis se inicia a partir do endereço 0x0100, como será estudado nesta seção.

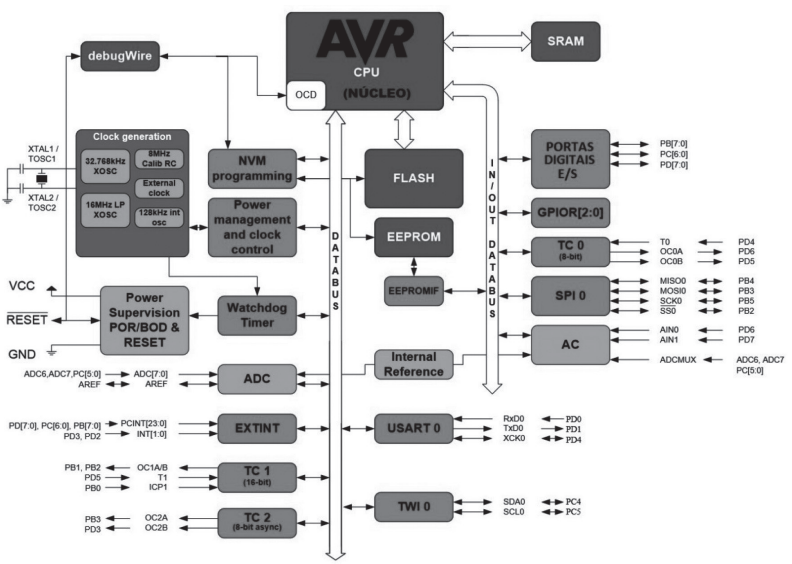
Dessa forma, esta seção foi escrita para você compreender a estrutura básica do microcontrolador ATmega328, com as principais informações sobre a sua construção, necessárias para a elaboração de projetos reais feitos nos próximos capítulos.

Para consolidar o seu estudo, retomaremos a situação-problema desta unidade: agora você é o responsável pelo projeto dos módulos eletrônicos internos de um novo modelo de carro. Todos os detalhes da fase desse projeto devem ser descritos em um relatório interno, usado para o catálogo de registro, como patrimônio intelectual, bem como para a aquisição de patentes. Sabendo que os módulos microprocessados serão construídos a partir do microcontrolador ATmega328, você deve registrar as operações mais básicas e a primeira que lhe foi incumbida é a simples soma de duas variáveis. Considerando a operação de soma de duas variáveis (valores na memória RAM) e o armazenamento do resultado em uma terceira variável, descreva qual é a relação e como atuam os elementos: CPU (ULA, registradores e unidade de controle), barramentos, memória de programa, memória de dados e contador de programa. Esse é apenas o começo do relatório interno que documentará a descrição de todas as operações presentes nos produtos da empresa.

Não pode faltar

Para colocar em prática todo o conhecimento adquirido durante este curso, foi escolhido como protagonista o microcontrolador ATmega328, presente na popular placa de desenvolvimento Arduino UNO e fabricado pela Atmel. Este microcontrolador é da família AVR, de 8 bits, de baixo consumo (*low power*), tecnologia CMOS – *Complementary Metal–Oxide–Semiconductor* (semicondutor complementar de óxido e metal) e construído sobre a arquitetura AVR RISC – *Reduced Instruction Set Computer* (computador com um conjunto reduzido de instruções), com arquitetura *Harvard* modificada. Pode alcançar uma velocidade de processamento de até 20 MIPS (milhões de instruções por segundo), operando em 20 MHz. Sua memória de programa (FLASH) é de 32 *kbytes*, possui 2 *kbytes* de memória RAM e 1 *kbyte* de EEPROM. Observe que sua arquitetura básica é exibida na Figura 1.5. Não se preocupe com os periféricos que não foram apresentados e com os termos em inglês, pois eles estão presentes apenas para referências no futuro, quando forem estudados, e serão devidamente descritos.

Figura 1.5 | Arquitetura interna do ATmega328



Fonte: <https://goo.gl/wTs9iD>. Acesso em: 29 abr. 2017.

É possível verificar na Figura 1.5 que estão presentes os principais periféricos internos, apresentados anteriormente na Seção 1.1 desta unidade, os quais serão estudados individualmente, começando pelo **núcleo**, ou **CPU** – *Central Processing Unit* (unidade central de processamento):

Núcleo

Apesar de não ter o papel de armazenar dados, algumas informações importantes ficam contidas em registradores internos ao núcleo. Os dados a serem processados pela ULA (Unidade Lógica e Aritmética), ou seus endereços de memória, ficam nos **Registradores de Propósito Geral**, enquanto informações para gerenciamento do processo ficam em outros registradores especiais, de propósitos específicos.

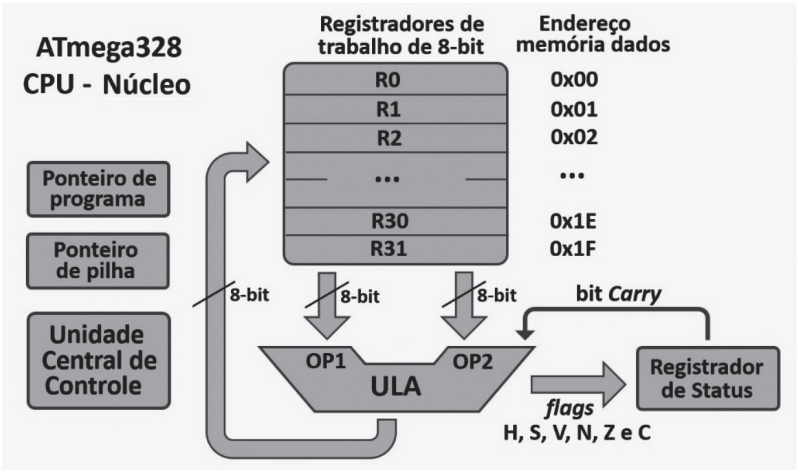
Devido à família de microcontroladores com o núcleo AVR, como o caso do ATmega328VR, possuir a arquitetura Harvard (modificada, pois há mais de um barramento de dados), as instruções são buscadas na memória de programa por um barramento exclusivo ao de dados e são executadas por um processo de canalização (*pipeline*) de nível unitário: enquanto uma instrução está sendo executada, a próxima já é pré-carregada do programa de memória, permitindo que estas sejam executadas continuamente, uma em cada ciclo de relógio ou *clock*.

Registradores de propósito geral

São 32 registradores de 8 bits que possuem exclusivamente acesso direto à ULA, permitindo que a maioria das operações seja feita em um único ciclo de clock. Portanto, todos os dados processados na ULA, tanto constantes na memória de programa, quanto variáveis na memória RAM, devem ser previamente transferidos para esses registradores, para então serem operados.

Na Figura 1.6 é possível ver o interior do núcleo. Vale ressaltar que os números mostrados com o prefixo "0x" estão representados na base hexadecimal, assim como é feito na maioria das linguagens de programação. As denominadas *flags*, ou bandeiras de indicação, servem para anunciar informações importantes sobre o processo em andamento e serão explicadas logo adiante, no registrador de status.

Figura 1.6 | Núcleo do ATmega328



Fonte: elaborada pelo autor.

Reflita

Os registradores de trabalho geral recebem esse nome porque são utilizados em todas as instruções do processador. As operações não podem ser feitas diretamente sobre os dados na memória, pois não há ligação direta. Dessa forma, todos os dados processados sempre passam por esses registradores e a maioria das instruções em um programa são de transferência interna de dados.

Seis dos registradores gerais podem ser agrupados em duplas, formando um trio de ponteiros de endereçamento indireto, de 16 bits, para acessar o espaço de dados, melhorando cálculos de endereçamento. Um destes três ponteiros de endereços pode também ser usado como ponteiro para recolher conteúdos na memória de programa (FLASH), como uma tabela de *look-up* ou tabela *hash*.

Pesquise mais

Este tipo de tabela não possui uma tradução oficial, recebe o nome de tabela de consulta (ou busca ou espalhamento), é muito utilizada por programadores de todas as áreas e nós a implementaremos futuramente.

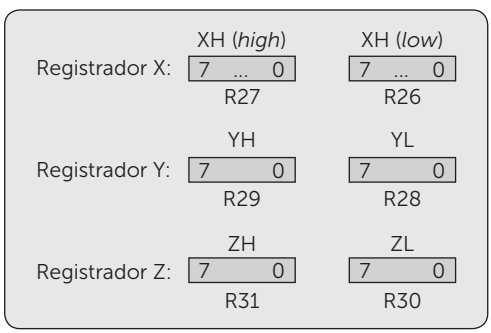
Uma tabela de senos pode ser criada para buscar os valores do seno a partir de seu índice, por exemplo. Entenda como esta técnica funciona em: <http://dcm.ffclrp.usp.br/~augusto/teaching/icii/Hash-Tables-Apresentacao.pdf> ou <https://www.ime.usp.br/~pf/mac0122-2002/aulas/hashing.html>. Acesso em: 9 jun. 2017.

Os registradores configurados dessa forma são denominados de registradores de 16 bits X, Y e Z. Conforme visto na Figura 1.7, cada registrador geral possui um endereço de memória, mapeando-os diretamente nos primeiros 32 locais do espaço de dados do usuário RAM. Apesar de não ser implementada fisicamente na memória, esta organização proporciona grande flexibilidade no acesso dos registradores, já que os registros de ponteiros X, Y e Z podem ser definidos para indexar qualquer registro no arquivo.

Os registradores X, Y e Z

Os registradores capazes de se agrupar são o R26 até o R31 e são definidos como descrito na figura:

Figura 1.7 | Registradores de 16-bit X, Y e Z



Fonte: elaborada pelo autor.

Nos diferentes modos de endereçamento, estes registradores de endereços têm funções como deslocamento fixo, incremento automático e decremento automático.

ULA

Conectada diretamente aos registradores gerais, a unidade ULA é responsável por executar operações entre estes, ou entre

um registrador e uma constante, de acordo com o comando decodificado da instrução correspondente, aplicado pela **unidade central de controle**. As operações podem também ser executadas com um único operando e são divididas em três categorias principais: aritmética, lógica e funções de bit. Após uma operação aritmética, o registrador de status é atualizado para exibir informações sobre o resultado da operação.

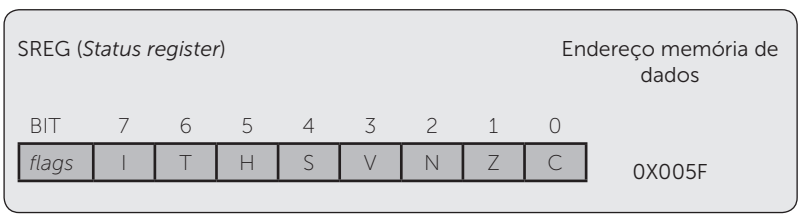
Algumas implementações da arquitetura AVR também fornecem um hardware multiplicador para valores com sinal e fracionários. Os seguintes esquemas de transferência interna ao UCP (unidade central de processamento) são suportados, sobre ponto de vista da ULA:

- Um operando de entrada de 8 bits e um resultado de saída de 8 bits.
- Dois operandos de entrada de 8 bits e um resultado de saída de 8 bits.
- Dois operandos de entrada de 8 bits e um resultado de saída de 16 bits.
- Um operando de entrada de 16 bits e um resultado de saída de 16 bits.

Registrador de status

Armazena informações sobre o resultado da instrução aritmética mais recentemente executada, que podem ser usadas para alterar o fluxo do programa através de operações condicionais. Quando o programa sofre o desvio para uma rotina de interrupção, este registrador deve ser armazenado na pilha, como será estudado, pois isso não é feito automaticamente via hardware. Conheceremos a descrição desse registrador especial.

Figura 1.8 | Registrador de status do ATmega328



Fonte: elaborada pelo autor.

- Bit 7 – I: *Global Interrupt Enable*: Bit de Ativação de Interrupção Global. Deve ser acionado para que as interrupções possam ocorrer. Além deste, as fontes de interrupção devem ser acionadas individualmente. Este bit funciona como um "disjuntor geral" das interrupções, quando objetiva-se desabilitar todas (individualmente habilitadas) de uma única vez; ele é automaticamente zerado por hardware quando ocorre uma interrupção e é reativado quando a rotina de interrupção termina, para permitir que outras rotinas de interrupção sejam tratadas, caso estejam pendentes.
- Bit 6 – T: *Copy Storage*: Bit Armazenamento de Cópia. Usado por instruções de cópia de bits.
- Bit 5 – H: *Half Carry Flag*: Indicador de meio carregamento. Aponta que ocorreu o fenômeno *Half care* em algumas operações aritméticas, como manipulação de dados na forma BCD – Binary-coded decimal (codificação binária decimal).
- Bit 4 – S: *Sign Bit*: Bit de sinalização. $S = N \oplus V$. É o resultado da operação "ou exclusivo" entre o bit indicador de negativo **N** e indicador de excesso em complemento de dois **V**.
- Bit 3 – V: *Two's Complement Overflow Flag*: Indicador de excesso em complemento de dois para operações aritméticas.
- Bit 2 – N: *Negative Flag*: Indicador de Negativo. Aponta se o resultado das operações é negativo, tanto lógicas quanto aritméticas.
- Bit 1 – Z: *Zero Flag*. Indicador de zero. Indica se o resultado de qualquer operação é zero. Muito usado em saltos condicionais.
- Bit 0 – C: *Carry Flag*. Indicador de excesso. Conhecida como o sinal de "vai um" dos somadores digitais. Mostra que o resultado de uma operação lógica ou aritmética excedeu o tamanho da palavra, de 8 bits.

Assimile

O registrador de status é muito importante para a execução do programa, pois todos os testes e desvios condicionais dependem das informações nele contidas.

Ponteiro de pilha

A pilha é usada principalmente para armazenar dados temporários, variáveis locais e endereços de retorno após interrupções e chamadas de sub-rotina. O sistema de pilha é implementado de maneira invertida, parte do endereço mais alto e assume endereços menores a partir do momento que a pilha aumenta. A única finalidade do ponteiro de pilha é apontar para o endereço do último dado armazenado na pilha e apenas este pode ser retirado. Dessa forma, os primeiros dados a serem colocados serão os últimos a serem tirados, seguindo uma fila do tipo LIFO ou FILO – First Input, Last Output (primeiro a entrar, último a sair).

A pilha deve ser definida na RAM pelo programa antes da ocorrência de qualquer sub-rotina ou interrupção. O valor do ponteiro de pilha inicial é igual ao último endereço da RAM e este registrador é implementado como dois registos de 8 bits no espaço de entradas e saídas. A pilha é efetivamente alocada na memória RAM e consequentemente o tamanho da pilha é limitado exclusivamente pelo tamanho de memória RAM, além de quanto dela será utilizado por outras finalidades.

Contador de programa

Funciona também como um ponteiro, porém indica para o processador qual é o endereço da próxima instrução a ser executada na memória de programa. É incrementado automaticamente à medida que as instruções são executadas, exceto quando ocorrem saltos, que têm seus endereços indicados na própria instrução. Estes saltos, diretos ou por chamadas de sub-rotina, determinam o fluxo do programa e podem desviá-lo para qualquer endereço válido. Também pode ser diretamente manipulado pelo programa usuário.

Unidade central de controle

Considerado o núcleo do núcleo. Esta máquina de estados é responsável por controlar de fato todos os outros componentes, executando as operações descritas pelas instruções, depois de decodificadas. Opera a partir do princípio da Máquina de Turin Universal, realizando as ações de busca, decodificação e execução.

Barramentos

O barramento de dados é de 8 bits, caracterizando o número de bits do microcontrolador. As instruções do ATmega328 são de 16

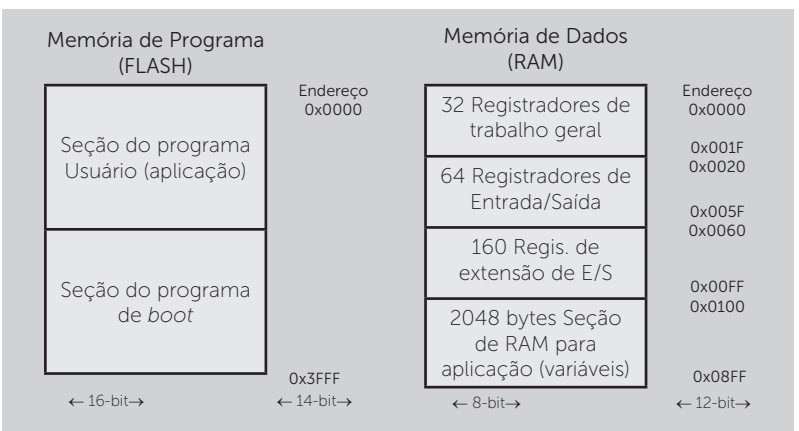
ou 32 bits (a maioria é de 16 bits). Assim, cada instrução consome dois ou quatro bytes na memória de programa (um *byte* par e um ímpar). O acesso às posições de memória, dado pelo contador de programa (*Program Counter* – PC), é realizado de dois em dois *bytes*, começando sempre por uma posição par. Portanto, o barramento de endereços deve ser capaz de endereçar sempre posições pares da memória de programa. Assim, o bit menos significativo do barramento de endereços pode ser desprezado. Desta forma, para a memória de 32 *kbytes* ($2^5 . 2^{10} = 2^{15}$ *bytes*) do Atmega328 são necessários 14 bits de endereçamento (2^{14} = 16.384 endereços) (LIMA; VILLAÇA, 2012).

Memórias e espaços

O espaço de memória de programa é dividido em duas seções: a parte de "*Boot Program*", ou programa de inicialização, e a parte de programa de aplicação, ou programa usuário. Ambas as seções possuem bit de travamento para proteções de escrita ou leitura e escrita.

Durante interrupções e chamadas de sub-rotinas, o endereço de retorno do PC é armazenado na pilha. Assim como o ponteiro de pilha, todos os programas usuários devem inicializar o SP na rotina de reset, antes de sub-rotinas e interrupções ocorrerem, ou seja, a primeira ação a ser feita quando ligado. Os dados na SRAM podem ser facilmente acessados através de cinco diferentes modos de endereçamento suportados pela arquitetura AVR. Os espaços de memória na arquitetura AVR são todos lineares e regulares.

Figura 1.9 | Espaços de memória de programa e de dados do ATmega328



Fonte: elaborada pelo autor.

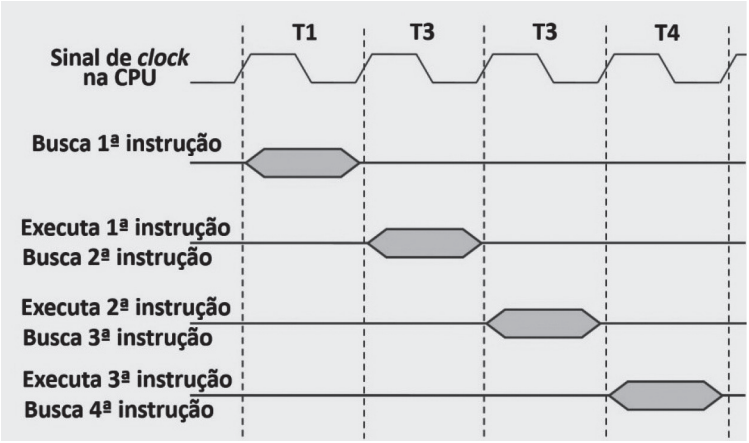
Tempo de execução de instruções

O sinal de *clock* usado pelo núcleo AVR é gerado diretamente pela fonte de relógio selecionada para chip e nenhuma divisão de relógio é feita internamente. O paralelismo gerado pelo sistema de *pipeline*, onde a busca e execução de instruções ocorrem simultaneamente, é permitido pela arquitetura Harvard e provê o conceito de rápido acesso ao arquivo de registros.

Exemplificando

Para entender como esse processo de *pipeline* funciona, observe a Figura 1.10, que mostra em cada ciclo de relógio uma instrução e executada e a outra seguinte já é capturada pela CPU para ser executada no ciclo seguinte.

Figura 1.10 | Execução das primeiras instruções em *pipeline*



Fonte: elaborada pelo autor.

Tratamento do reset e interrupções

O sistema AVR fornece várias fontes de interrupção diferentes. Assim como o vetor de reset, essas interrupções possuem vetores de programa individuais, que apontam, cada um, para sua respectiva rotina no espaço de programa. Isso significa que, assim que uma interrupção ocorre, o programa salta para o endereço indicado no vetor dessa interrupção, onde fica a devida rotina de tratamento, e

retorna para onde estava antes do desvio. Todas as interrupções são ativadas através de bits individuais, nos devidos registradores, além do bit habilitador global de interrupções, visto no registrador de status.

Pesquise mais

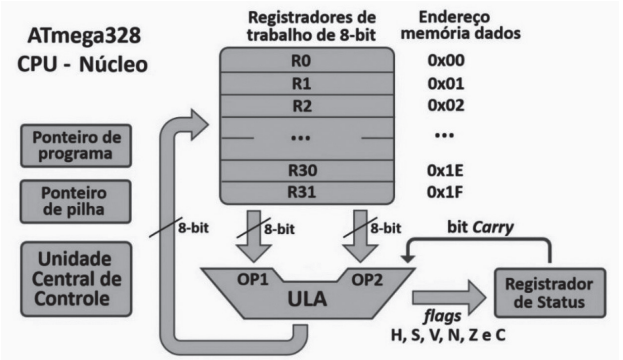
O sistema de interrupções e reset do ATmega328P possui mais detalhes interessantes de serem observados. Leia o curto trecho a respeito no seu *datasheet*, ou manual, na página 32: <http://www.atmel.com/pt/br/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso em: 10 jun. 2017.

Sem medo de errar

Uma montadora de veículos está na fase de criação de um novo modelo e, como esta trabalha com muitos serviços terceirizados, contratou algumas pequenas empresas de desenvolvimento tecnológico para construir as diversas partes do novo modelo. Nesse contexto, você é funcionário da instituição responsável por projetar os módulos eletrônicos internos de toda a parte elétrica do carro. Todos os detalhes da fase de projeto devem ser descritos em um relatório interno, usado para o catálogo de registro, como patrimônio intelectual, bem como para a aquisição de patentes. Sendo assim, você deve documentar as primeiras operações, as mais básicas, descrevendo o processo que ocorre internamente ao microcontrolador, ou seja, qual é a relação e como atuam os elementos: CPU (ULA, registradores e unidade de controle), barramentos, memória de programa, memória de dados e contador de programa, considerando a operação de soma entre dois valores na memória RAM e a devolução do resultado.

O contador de programa (PC), ou ponteiro de programa, que é um registrador especial dentro do núcleo (ver Figura 1.11), aponta para instruções usadas para esta tarefa na memória de programa. Isso é feito diretamente através do barramento de endereço de instruções (oculto na Figura 1.12, mas presente na Figura 1.13). Pelo mecanismo interno, assim que o valor do PC é atualizado, esse valor é transmitido pelo barramento de endereço de instruções e as instruções correspondentes aos valores do PC são transferidas para o processador (núcleo) através do barramento de instruções, indicado pelo círculo número 1 na Figura 1.12.

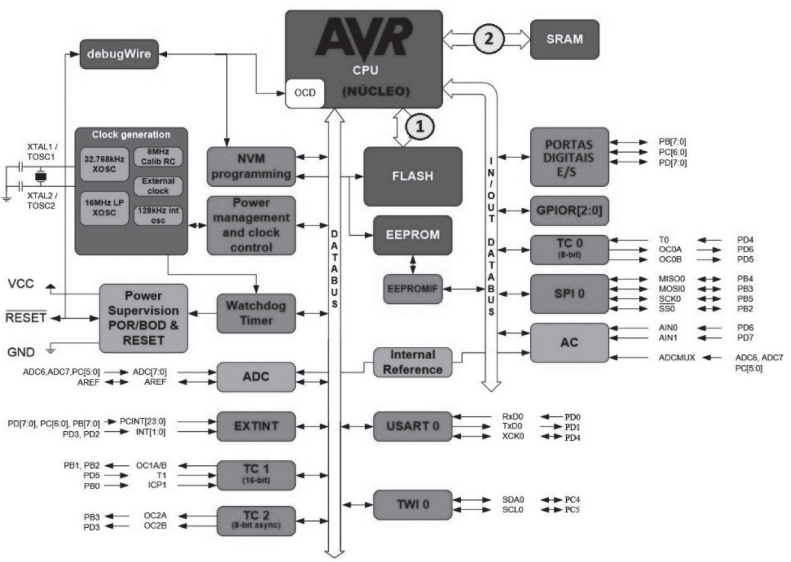
Figura 1.11 | Núcleo do ATmega328



Fonte: elaborada pelo autor.

A informação é decodificada e executada efetivamente pela unidade central de controle (UCC), que age autonomamente sobre as partes internas à CPU, como os barramentos de endereços. Para o nosso caso, como serão buscados valores na RAM, a UCC coloca o correspondente endereço no barramento de endereço de dados (também oculto na Figura 1.12, mas presente na Figura 1.13) e no ciclo de relógio seguinte busca o valor daquele endereço no barramento de dados, indicado pelo círculo número 2, da Figura 1.12.

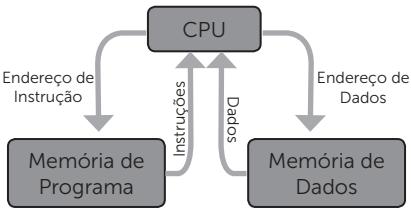
Figura 1.12 | Estrutura interna do microcontrolador ATmega328



Fonte: adaptada de <https://goo.gl/QH9wtt>. Acesso em: 29 abr. 2017.

De acordo com a instrução, a UCC colocará os valores das duas variáveis buscadas na RAM nos respectivos registradores de propósito geral e, depois disso, realizará a soma desses dois operadores, controlando quais registradores são conectados à ULA e indicando a esta que a operação a ser feita é uma soma. De maneira automática, a ULA informa ao registrador de status o valor de todas as flags afetadas pelo resultado desta última operação.

Figura 1.13 | Barramentos da arquitetura Harvard do ATmega328



Fonte: elaborada pelo autor.

Dessa forma, é necessário um total de quatro ações para completar essa tarefa computacional:

1. O primeiro operando é carregado da memória de dados para um dos registradores gerais.
2. O segundo operando é carregado da memória de dados para algum outro dos registradores gerais.
3. Os dois valores são processados (somados) pela ULA e o resultado é retornado para um dos dois registradores gerais usados.

O resultado é transferido para o devido endereço da memória de dados.

Avançando na prática

Definição de uma arquitetura de computador

Descrição da situação-problema

Uma empresa de microcontroladores está desenvolvendo um novo modelo e contratou você como engenheiro de arquitetura de computadores para definir como será a estrutura interna do

conjunto. Para limitar o preço e a utilidade do novo chip, a empresa usou os seguintes requisitos:

- O processador é de 16 bits.
- As instruções são todas de 32 bits.
- A memória de programa (FLASH) tem 32 *kbytes*.
- A memória RAM (ou memória de dados temporários) tem 4 *kbytes*.
- Ambas as memórias têm largura de 8 bits, ou seja, um *byte* em cada endereço.

Sabendo que esta será uma máquina Harvard, defina quais devem ser as larguras dos quatro barramentos: endereço de instrução, instrução, endereço de dados e dados.

Resolução da situação-problema

Considerando que o processador é de 16 bits, significa que este processa dois operadores de 16 bits em cada ciclo de operaçãos e, portanto, o barramento de dados deve ser de 16 bits de largura. Sabendo que as instruções são todas de 32 bits, é necessário que o barramento de instruções seja de também 32 bits, para que o processador possa buscar uma instrução completa em cada ciclo de relógio. Foi dito que as memórias possuem 8 bits de informação em cada endereço, portanto são necessários 12 bits para endereçar os 4 *kbytes* de memória de dados, pois com 12 bits é possível gerar 4096 ($=2^{12}$) números diferentes, ou seja, endereços de memória. Esse número é igual a 4*K*, pois $4k = 4^* 2^{10} = 2^{12}$. Isso define a largura de 12-bits para o barramento de endereço de dados. Se as instruções fossem também de 8 bits, a conta seria a mesma para o barramento de endereço de instruções, pois cada instrução estaria em um endereço de memória. Porém, cada instrução ocupa exatamente quatro posições de memória, uma vez que o enunciado diz que as instruções são todas de 32-bits, o que significa que a primeira instrução estará no byte de endereço zero e a segunda instrução estará no *byte* de endereço 4. Como esse barramento de endereços acessará somente endereços múltiplos de 4, os últimos dois bits menos significativos não precisam ser considerados e podem ser desprezados pelo sistema.

Desse modo, os 32 *kbytes* de memória de programa, que precisariam de 15 bits para acessar todos os endereços em *bytes* ($32k = 2^5 * 2^{10} = 2^{15}$), precisarão de apenas 13 bits para acessar apenas endereços múltiplos de 4. Isso define o barramento de endereço de instruções de 13 bits. Esse mesmo valor pode ser obtido por um outro raciocínio equivalente: Se cada "andar" da memória FLASH possuísse 32 bits de largura, seriam necessários 32k/4 = 8k andares para alocar os 32 *kbytes* de memória de programa, ou seja, 8k endereços. Assim, são necessários 13 bits para gerar 8K endereços diferentes, pois $8K = 8* 2^{10} = 2^{13}$.

Faça valer a pena

1. O núcleo possui um pequeno espaço de memória volátil, onde se encontram os 32 registradores de propósito geral, ou registradores de trabalho, muito utilizados durante a execução do programa embarcado.
Os registradores de trabalho, ou registradores de propósito geral, são usados para:

a) Armazenar os resultados dos processamentos por tempo indefinido para futuras decisões.
b) Calcular operações lógicas e aritméticas e transferir os resultados para a ULA.
c) Guardar instruções do programa que podem ser utilizadas em qualquer momento.
d) Armazenar os valores que serão processados pela ULA, pois têm acesso direto e exclusivo.
e) Receber as instruções das rotinas de interrupção para paralelizar o tratamento.

2. Apesar do núcleo do microcontrolador não possuir o papel de armazenar massas de dados, algumas informações importantes ficam temporariamente retidas, ou nos registradores de propósito geral, ou em alguns registradores especiais, como os ponteiros de programa e de pilha e o registrador de status.
As informações contidas no registrador de status são importantes para o programa porque:

a) Refletem como foi o resultado da última operação, decidindo e desviando o fluxo do programa através de operações condicionais.
b) Auxiliam o programa a decidir se as variáveis serão guardadas na memória RAM ou na memória EEPROM.

c) Informam através de suas flags quais foram as últimas cinco rotinas ou funções chamadas pelo programa principal, útil para o programa situar sobre os caminhos percorridos no fluxograma.
d) Representam os dados mais importantes e que devem ficar armazenados para processamentos futuros.
e) São os valores das constantes usadas pelo programa atual que devem ficar no núcleo para rápido acesso.

3. A arquitetura de qualquer microcontrolador é caracterizada, entre outas coisas, pelas dimensões ou larguras dos barramentos internos, como são as conexões entre o núcleo, memórias e periféricos internos e o tamanho das memórias. No microcontrolador Atmega328, esses parâmetros foram escolhidos para prover uma estrutura eficiente para o conjunto de recursos disponíveis.
A respeito da arquitetura e barramento do ATmega328P, considere as seguintes afirmações:

I – Os endereços das memórias são de 8 bits e armazenam dados de 16 ou 32 bits, o que define o número de bits do processador.
II – O contador de programa (PC) só pode assumir valores pares, pois as memórias armazenam dados de 8 bits em cada endereço e as instruções são todas de 16 ou 32 bits.
III – As memórias de dados e de programa possuem barramentos diferentes, tanto para seus conteúdos quanto endereços, pois a arquitetura adotada por esse chip é do tipo Harvard.
IV – As instruções são de 8 bits e manipulam dados que podem ser de 8, 16 ou 32 bits, dependendo da necessidade de processamento.
Considerando a sequência, qual das alternativas abaixo retrata corretamente quais afirmações são falsas e quais são verdadeiras?

a) F, V, F, V.
b) F, F, V, F.
c) F, V, V, F.
d) V, V, F, F.
e) F, V, V, V.

Seção 1.3

Linguagem *Assembly*

Diálogo aberto

Um entendimento aprofundado sobre o funcionamento dos microprocessadores transcorre na programação em sua forma mais fundamental: instrução por instrução. E para isso, antes de começar a construir códigos em *Assembly*, um programador deve conhecer intimamente os recursos da máquina que estará operando. Você perceberá que usaremos diretamente os conceitos de arquitetura do AVR, estudados na segunda seção desta unidade. Agora, você entenderá a importância desses conceitos e a utilização deles para a programação de sistemas embarcados. Depois de estudar essa nova seção e de compreender o exemplo final, você estará pronto para criar novos programas em *Assembly*.

Estudaremos aqui os principais aspectos das instruções utilizadas na programação em *Assembly*, para os microcontroladores AVR, como o ATmega328. Para colocar esse conhecimento em prática, resolveremos as últimas questões sobre a situação-problema apresentada para esta unidade. Lembre-se de que uma empresa montadora de veículos está na fase de criação de um novo modelo e contratou você como profissional responsável para projetar os módulos eletrônicos internos desse carro. Todos os detalhes da fase de projeto devem ser descritos em um relatório interno, usado para o catálogo de registro, como patrimônio intelectual, bem como para a aquisição de patentes, como descrito na seção anterior.

O próximo trabalho será registrar as etapas mais básicas do processamento executado nesses módulos, porém agora em termos de programação e não de funcionamento interno do sistema. Dessa forma, você foi incumbido de escrever o trecho de código correspondente à ação de: buscar os valores contidos nos endereços 0x0109 e 0x010A para os registradores R16 e R17, somá-los e devolver o resultado para o endereço 0x010B. Além disso, você deve descrever como o processador atua para executar cada instrução e quais são as consequências indicadas pelo registrados de

status. Toda a teoria necessária estará nesta seção. Vamos colocar todo esse conhecimento em prática?

Não pode faltar

Linguagem de máquina para o microcontrolador AVR

O desenvolvimento de projetos em linguagem de máquina é praticamente inexistente por se tratar de um trabalho lento e tedioso, todavia ainda é possível encontrar várias ferramentas que auxiliam a desenvolver este tipo de trabalho. Já o *Assembly* é uma linguagem que está um nível acima de abstração, nas camadas de abstração, quando comparada com a linguagem de máquina, pois suas instruções são mnemônicas (rótulos ou apelidos) que correspondem individualmente a comandos (ou instruções) em linguagem de máquina. Existem, ainda, linguagens de mais alto nível, como C, que para tornar seus programas embarcáveis na memória devem ser convertidos por um compilador para um código em *Assembly* ou diretamente em linguagem de máquina. Vale salientar que as instruções em linguagens de mais alto nível geralmente correspondem a muitas instruções em linguagem *Assembly* e isso permite a construção em blocos padrões, o que agiliza muito o processo de programação.

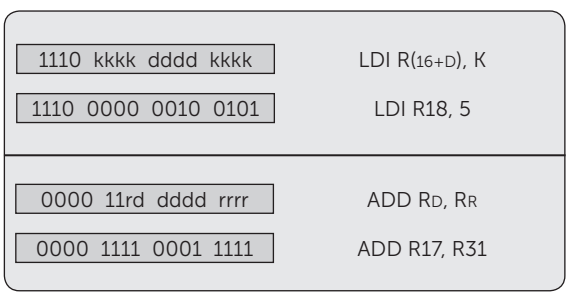
A programação em linguagem de máquina (literalmente escrever zeros e uns) exige um entendimento técnico do programador de como as instruções são representadas, formadas, escolhidas e executadas. Todas as instruções computacionais seguem um padrão básico: possuem um prefixo que identifica a instrução, ou seja, qual é a ação que deve ser executada pelo processador. Os bits desse prefixo são chamados de **bits de controle** e indicam se aquela instrução é uma soma, uma multiplicação, uma operação lógica, uma transferência de dados ou um salto no programa. Seguido desse prefixo, na instrução existe mais um ou dois campos, referentes aos operadores, ou seja, quais elementos a serem processados na execução da instrução. Nesses campos são identificados: valores constantes, registradores de trabalho e endereços da memória a serem acessados. Os bits nesse campo são chamados de **bits de dados**.

Na criação de códigos em linguagem de máquina, o programador deve realizar operações aritméticas a próprio punho para determinar

alguns valores na instrução, além de escolher individualmente os bits de controle. A alteração de uma parte do programa pode causar efeitos em outras instruções do código, o que demanda algumas análises detalhadas do programa cada vez que este sofre alguma edição. Porém, apesar da programação nessa forma ser algo realmente trabalhoso e propenso a erros, é importante apresentá-la inicialmente para que estudantes, como você, possam entender como os programas embarcados são construídos e executados. Fique tranquilo porque não construiremos códigos assim, apenas abordaremos um pequeno exemplo para demonstrar como é construído um programa na sua forma mais crua: apenas zeros e uns. Nesse trecho de código, somaremos 1 + 2 e, como aprendido anteriormente, todos os valores a serem operados pela ULA devem estar nos registradores de trabalho (dentro do núcleo). Para isso é usada a instrução LDI, que será estudada mais adiante ainda nesta seção, e possui o prefixo indicativo "1110" (0xE em hexadecimal).

Dessa forma, essa tarefa deve ser composta por três instruções, duas para armazenar os valores 1 e 2 em dois registradores de trabalho e uma para realizar a soma e guardar o resultado, a instrução ADD, de prefixo "000011". Essas instruções são de 16 bits e têm o formato apresentado na Figura 1.14. Nessa nomenclatura, Rd significa registrador de destino, ou seja, onde é armazenado o resultado da instrução, além de ser o primeiro operando em instruções lógicas e aritméticas, como será visto logo adiante. O segundo operador, dependendo da instrução, pode ser o K ou o Rd, que corresponde a uma constante e um registrador de origem, respectivamente.

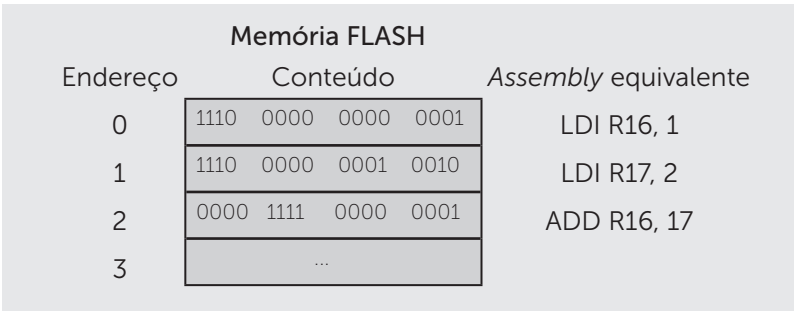
Figura 1.14 | Formato das instruções LDI e ADD



Fonte: elaborada pelo autor.

Perceba que os campos não são contínuos e o programador deve encontrar o valor do campo D=2 pela solução 16+D=18, pois para essa instrução o primeiro endereço (zero) corresponde ao registrador R16. Portanto, considerando que este código seja a primeira parte do programa, ou seja, está no início da memória FLASH, ela estaria com os seguintes valores nos primeiros endereços, apresentados na Figura 1.15:

Figura 1.15 | Valores contidos no início da memória FLASH



Fonte: elaborada pelo autor.

Assembly

Realmente a linguagem de máquina não é uma maneira agradável de criar um programa. Assim, agora estudaremos como construir um programa em linguagem *Assembly*, bem próxima da linguagem de máquina, pois tem uma relação direta, um a um, para todas as instruções. No *Assembly*, as instruções de máquina recebem mnemônicos, ou seja, apelidos, abstraindo o programador de alguns detalhes e aumentando a legibilidade do código. Cada linha corresponde a uma instrução e segue o seguinte modelo:

[Rótulo] [Comando] Rd, k [;comentário]

O rótulo representa um apelido para o endereço da memória onde estará essa instrução, pode não existir e é usado pelo programa como referência para as instruções de salto, que veremos mais à frente, nesta seção. O comando é onde está o mnemônico, ou seja, a instrução que o núcleo deve executar. O **Rd** representa o registrador de destino, ou seja, qual dos registradores de propósito geral receberá o resultado da instrução. Para instruções de deslocamento de dados para a memória, o Rd assume um endereço

de memória, para onde o dado deve ser transferido, como será visto adiante. O **K** é o registrador de origem ou a constante usada na operação. Apesar de ser minoria, algumas instruções são de apenas um operador, representado por Rd. O comentário deve começar com o ";" (ponto e vírgula) e não causa nenhum efeito no programa. Apesar de inicialmente parecer sem utilidade, é muito importante o hábito de comentar os programas enquanto são construídos, para que outras pessoas possam entender mais facilmente ou até mesmo o próprio programador, quando for rever o programa. Os comentários são facultativos.

Estrutura de um programa embarcado

Os programas embarcados possuem uma estrutura básica específica, que os diferenciam dos programas tradicionais de alto nível (para computadores) basicamente por não serem encerrados. Assim, como os sistemas operacionais, os programas embarcados devem estar sempre em atividade, prontos para operar, até que o sistema seja desligado. No entanto, existem algumas técnicas de hibernação que permitem o "congelamento" do processamento para poupar energia. Os programas embarcados podem ser construídos a partir de inúmeros fluxogramas diferentes, mas todos devem respeitar uma estrutura básica, apresentada na Figura 1.16:

Figura 1.16 | Estrutura básica de programas para sistemas embarcados

Fonte: elaborada pelo autor.

Podemos observar que a parte da inicialização ocorre apenas uma vez, quando o programa é iniciado. Em seguida, o sistema entra em seu ciclo de operação, onde realizará o seu trabalho.

Assimile

Na fase de inicialização, é feita toda a configuração do sistema, preparando o conjunto para os trabalhos que serão realizados. Essas primeiras ações são: definição de quais pinos do microcontrolador

serão entradas ou saídas; configuração dos periféricos e definição de como eles serão utilizados; declaração das variáveis usadas pela função principal ou variáveis globais, chamadas de funções ou rotinas que devem ser executadas apenas no início do processo.

Deve ficar claro que o código pertencente a essa fase é executado apenas uma vez, tendo essa ação repetida apenas se o microcontrolador for reiniciado. A partir disto, o sistema está pronto para iniciar sua fase de trabalho propriamente dita, etapa que recebe o nome de loop infinito, pois é onde o programa deve permanecer enquanto o dispositivo estiver energizado. Basicamente, as ações sequências nesse laço são: leitura das entradas, processamento dos dados e atualização das saídas.

Diretivas de *Assembly*

Enquanto as instruções informam à CPU o que ela deve fazer, as diretivas de pré-compilação (também chamadas de pseudoinstruções) são usadas pelo compilador no momento de gerar o programa em linguagem de máquina. Esse tipo de comando não se torna instruções executadas, mas cria amarrações que ajudam muito na fase de programação. Por exemplo, você pode definir a constante MEIADUZIA = 6 e usar esse nome no programa. Na hora da compilação, todos os nomes são antes substituídos pelo seu valor definido. As principais são:

- **.EQU** (*equate*), diretiva usada para definir constantes, como a tradicional diretiva "#define", utilizada por muitas linguagens, como a C (MIZRAHI, 2008) e também define endereços fixos da memória de dados, ou seja, as variáveis do programa. Desse modo, se um programa monitora o sinal de temperatura, por exemplo, e armazena o seu valor sempre no endereço 0x021A, a diretiva pode ser usada: .EQU TEMPERATURA = 0x021A. Ou, para definir uma constante, de nome MEIADUZIA, por exemplo, teríamos: .EQU MEIADUZIA = 6.

Você pode perceber que para os dois casos, o comando é igual, o que diferencia é a forma com que a definição é usada, ou melhor, se é empregada no lugar de um valor ou de um endereço.

- **.SET** (*equate*), diretiva que funciona da mesma maneira que a .EQU, com a diferença de poder ser modificada

posteriormente no programa, assumindo um novo valor dali em diante.

- **.ORG** XX (*origin*), diretiva usada para indicar, no programa escrito, um endereço forçado para as instruções abaixo, ditado pelo valor XX. É sempre empregado para definir onde será o início do endereçamento, com XX = 0, ou seja, que a instrução abaixo dele será a primeira e estará no endereço 0x0000. Serve também para definir endereços na memória de dados.
- **.INCLUDE** "NomeArquivo", avisa ao compilador que o arquivo indicado deve ser incluído e seu conteúdo considerado na compilação. Este arquivo pode conter outras rotinas usadas pelo programa ou apenas um conjunto de definições, como o arquivo que será usado em todos os nossos programas em *Assembly*, na primeira linha, a diretiva .INCLUDE "m32def.inc", permitindo o uso de nomes dos registradores no programa.

Exemplificando

Como foi mencionado, o *Assembly*, apesar de ser uma linguagem de baixo nível, traz consigo algumas facilidades, como o uso de definições. Para exemplificar como as diretivas de compilação funcionam, considere a instrução: **IN R12, PINB**. No lugar de PINB, o compilador insere o seu endereço correspondente na hora de gerar o código, que é 0x0018. Tanto essa quanto todas as outras definições para essa família de microcontroladores está presente no arquivo que deve ser incluído em todos os programas em *Assembly*: "m32def.inc".

Representação dos formatos de dados

Apesar de todos os dados serem armazenados no sistema na forma de bits (ou sequência de bits), existem outras formas de representar valores na escrita do programa.

- Números binários: são sequencias de bits iniciadas pelo prefixo 0b, portanto, se você quiser carregar o valor 5 no registrador R16, deve usar: **LDI R16, 0b00000101.**
- Números hexadecimais: algarismos hexadecimais (do 0 ao 9 e do A ao F) com o prefixo 0x. Assim, para armazenar o valor 17 no registrador R21, o comando é: **LDI R21, 0x11.**

- Números decimais: são os números da forma que usamos naturalmente e sem nenhum prefixo. A instrução para armazenar o valor 150 no registrador R18 é: **LDI R18, 150.**
- Caractere ASCII: também é possível manipular caracteres seguindo o padrão da popular tabela ASCII, com o seu valor binário correspondente. Para isso, ao invés de prefixo, usamos aspas simples. Para armazenar o caractere 'a' (0x61) no registrador R30 a instrução é: **LDI R30, 'a'.**

Conjunto de instruções AVR

Como foi visto na seção anterior, diferente da memória de dados, que possui 8 bits de informação em cada endereço, a memória de programa possui o dobro: 16 bits. Dessa forma, como a maioria das instruções são de 16 bits (algumas têm 32), o endereço 0x0000 corresponde à primeira instrução e o 0x0001 à segunda, assim sucessivamente. Pode-se perceber que uma instrução de 32 bits ocupará dois andares (endereços) da memória de programa

Estudaremos agora as principais instruções do conjunto, divididas em cinco grupos: lógicas e aritméticas; de salto; de transferência de dados; de bit e de teste de bit; de controle do microcontrolador. Pensando na didática, as instruções não serão apresentadas em seus grupos, mas em uma sequência voltada para a construção dos primeiros trechos de programa.

- **LDI** – ***L**oa**D** **I**mmediate* (carregamento imediato): esta instrução basicamente copia o valor imediato presente na instrução para algum dos registradores de propósito geral e tem o seguinte formato: LDI Rd, k.

O valor k deve ser de 8 bits, ou seja, de 0 até 255, e apenas a segunda metade dos registradores podem ser usados (R16 ao R32). Como já foi visto, para armazenar o valor 0x25 (ou 37 em decimal) no registrador R18, deve-se usar a expressão LDI R18, 0x25.

- **LDS** – ***L**oa**D** direct from data Space* (carregamento direto do espaço de dados): esta instrução armazena no registrador Rd o conteúdo do endereço de dado K, que pode ser de 0x0000 até 0xFFFF. Depois dessa instrução, as posições da memória de dados Rd e K terão o mesmo conteúdo e ela é usada quando se deseja buscar o valor de variáveis para processamento.
- **STS** – *Store direct to data Space* (armazenamento direto no

espaço de dados): basicamente, é o papel inverso da instrução anterior, ou seja, armazena no endereço K o conteúdo do registrador Rd no local na memória.

- **IN** – *IN from I/O location* (entrada de local de entrada/saída): como foi visto na segunda seção desta unidade, o ATmega328 possui um espaço de memória de I/O, usada para compartilhar dados entre os periféricos e o núcleo. Esse espaço é de fato uma região da memória de dados (RAM), porém com endereçamento diferente, usado em instruções de I/O, como a IN: IN Rd, A.

O papel desta instrução é trazer o conteúdo do endereço de I/O representado por A (de 0 até 63) para o registrador Rd (de 0 a 31), para então ser tratado pelo programa. Como será visto adiante, se um botão estiver conectado ao sistema pelo periférico PORTB (uma das portas digitais), o seu estado pode ser verificado pelo programa ao trazer o valor do registrador de I/O PINB para algum registrador, o R12, por exemplo. IN R12, PINB.

Devemos perceber que a instrução IN pode ser usada com o mesmo propósito da instrução LDS. No entanto, a instrução de I/O possui as seguintes vantagens: é executada mais rapidamente, pois possui 16 bits, contra 32 da LDS, além de permitir o uso dos nomes dos registradores de I/O, como no exemplo acima do PINB.

- **OUT** – *OUT to I/O location* (saída para local de entrada/saída): realiza troca de dados entre o programa e os periféricos internos, assim como a anterior, mas de maneira invertida. Serve para transmitir informação para o exterior, ou seja, controlar as saídas do sistema. Dessa maneira, se um LED estiver conectado ao conjunto através do periférico PORTC, ele pode ser acionado com os comandos: LDI R16, 0xFF OUT PORTC, R16.
- **MOV** – *MOVe instruction* (instrução de movimento): serve para transferir o conteúdo entre os registradores gerais. Por exemplo: MOV R10, R22; R10 ← R22.
- **TST** – *TeST instruction* (instrução de teste): testa o operando para a próxima instrução, que deve ser de salto, condicionado ao resultado deste teste (compara se o valor é igual a zero).
- **COM** – *COMplement instruction* (instrução de complemento): inverte os bits de seu único operando.

- **INC** – *INCrement instruction* (instrução de incremento): incrementa o seu único operador. Exemplo: INC R2 ; R2 ← R2 +1. Vale salientar que se R2 estiver no seu valor máximo, o incremento irá zerá-lo.
- **DEC** – *DECrement instruction* (instrução de decremento). Decrementa seu único operador. Usando o mesmo princípio da instrução INC, a instrução DEC faz uma variável nula assumir o máximo: 255.
- **ADD** – *ADD Instruction* (instrução de adição): adiciona dois operadores e guarda a soma no primeiro.

Instruções de salto: veremos agora as instruções de salto, sendo algumas delas condicionais, ou seja, sua ocorrência depende de algum resultado usado como critério de decisão.

- **JMP** – *JuMP instruction* – instrução de salto. Nesta instrução, não há operadores, apenas o rótulo indicando para onde o programa deve ser desviado, de maneira incondicional.
- **BREQ** – *BRanch if EQual* (salte se for igual): assim como a anterior, não possui operandos, apenas o rótulo para onde o programa deve ser desviado, caso a flag ZERO do registrador de status esteja acionada. Essa instrução, portanto, atua em conjunto com a instrução anterior, que deve ser aritmética (exceto pela **TST**) e que desvia o programa caso o seu resultado seja nulo (zero). Se for desejado um salto caso o resultado não seja nulo, usa-se a instrução inversa BRNE.
- **CALL** – Instrução de chamada de sub-rotinas: também não possui operandos, apenas o rótulo, e atua de maneira semelhante à JMP, com exceção de retornar ao lugar no programa onde foi chamada. Isso ocorre quando se executa a instrução **RET** (retorno), encerrando aquela sub-rotina.

Reflita

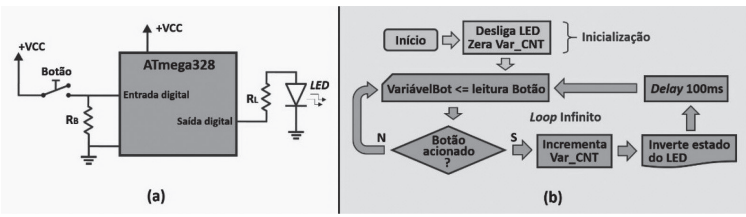
À primeira vista, o que aprendemos nesta seção pode não parecer muita coisa, mas, se você usar sua capacidade de criar algoritmos, verá que pode construir qualquer programa em *Assembly* a partir de um fluxograma, pois aqui estão todas as informações necessárias para isso.

As instruções de salto condicional que ditam os caminhos que o programa seguirá no fluxograma.

Exemplificando

Para aplicar essas instruções estudadas, consideraremos o simples exemplo: construir um algoritmo e um programa em *Assembly* para inverter o estado de um LED (inicialmente apagado) cada vez que um botão for pressionado, além de armazenar quantas vezes o botão foi acionado desde a inicialização do sistema. Para isso, abstrairemos inicialmente como as portas digitais são manipuladas e consideraremos que o botão está conectado em algum canal da porta A e o LED em algum canal (pino) da porta B. Um atraso deve ser utilizado a cada vez que o botão é acionado, para não contabilizar várias vezes. Um simples circuito e um possível fluxograma/algoritmo capaz de resolver o problema é mostrado na Figura 1.17.

Figura 1.17 | (a) Circuito; (b) fluxograma/algoritmo para o exemplo



Fonte: elaborada pelo autor.

O programa em *Assembly* correspondente pode ser visto na Figura 1.18.

Figura 1.18 | Programa em *Assembly* para o exemplo

```
.INCLUDE  "M32DEF.inc"      ;inclui nomes dos regs
.EQU  VAR_CNT = 0x120       ;reserva end. de dados

.ORG 0                      ;indica início do código
   LDI     R16, 0xFF
   OUT     DDRA, R16        ;config. PortA entrada
   LDI     R16, 0x00
   OUT     DDRB, R16        ;config. PortB saída
   OUT     PORTB, R16       ;desliga LED
   STS     VAR_CNT, R16     ;zera variável
LOOPINFINITO:
   IN      R16, PINA        ;leitura do botão
   TST     R16              ;testa valor
   BREQ    LOOPINFINITO     ;se não estiver acionado
   LDS     R16, VAR_CNT     ;lê valor do contador
   INC     R16              ;incrementa valor
   STS     VAR_CNT, R16     ;devolve valor atualizado
   IN      R16, PORTB       ;lê estado do LED
   COM     R16              ;inverte valor
   OUT     PORTB, R16       ;inverte estado do LED
   CALL    DELAY            ;invoca rotina de atraso
   JMP     LOOPINFINITO     ;retorna ao trabalho
DELAY:
   LDI     R23, 10
L3: LDI    R22, 100         ;carrega regs de contagem
L2: LDI    R21, 0xFF
L1: NOP                     ;não faz nada
   DEC     R21
   BRNE    L1          ;eqto R21 não zerar vai p L1
   DEC     R22
   BRNE    L2               ;idem para R22
   DEC     R23
   BRNE    L3               ;idem para R23
   RET                      ;retorna da rotina
```

Fonte: elaborada pelo autor.

O que vemos até aqui são as principais instruções, suficientes para fazer programas mais simples.

Pesquise mais

Infelizmente não conseguimos abordar todas as instruções aqui. Vale apenas usar um pouco do seu tempo para ver o resumo do conjunto de instruções oficial da AVR. Disponível em: <http://www.atmel.com/pt/br/Images/Atmel-0856-AVR-Instruction-Set-Manual.pdf>. Acesso em: 22 abr. 2017.

Sem medo de errar

Nesta unidade, estudamos as instruções e programação em *Assembly* para o microcontrolador da AVR ATmega328. Para usar nosso estudo na prática, resolveremos agora as últimas questões sobre a situação-problema apresentada para esta unidade: lembre-se de que você é o responsável por projetar os módulos eletrônicos internos de toda a parte elétrica do carro de uma montadora que está na fase de criação de um novo modelo. Todos os detalhes da fase de projeto devem ser descritos em um relatório interno, que comporá o catálogo de registro, usado como patrimônio intelectual, bem como para a aquisição de patentes. Você já definiu quais módulos serão microprocessados e já registrou algumas operações básicas em seu relatório, como a descrição de como o processador e as memórias internas devem se comunicar através dos barramentos para realizar uma soma de duas variáveis e devolver o resultado para a memória. Agora nesta etapa, o seu trabalho será novamente registrar um dos processos mais básicos do processamento executado nesses módulos, assim como a anterior, porém, agora será em termos de programação, e não de funcionamento interno do sistema. Dessa forma, você foi incumbido de escrever o trecho de código correspondente à ação de: buscar os valores contidos nos endereços 0x0109 e 0x010A para os registradores R16 e R17, somá-los e devolver o resultado para o endereço 0x010B. Além disso, você deve descrever como o processador atua para executar cada instrução e quais são as consequências indicadas pelo registrados de status. Apesar de se tratar de um pequeno trecho, escreveremos um programa completo que está descrito na Figura 1.19:

Figura 1.19 | Programa em *Assembly* para a situação-problema

```
.INCLUDE  "M32DEF.inc"      ;inclui nomes dos regs
.EQU  OP1 = 0x109           ;reserva end. dados para variável
.EQU  OP2 = 0x10A
.EQU  RES = 0x10B

.ORG 0                      ;indica início do código
   LDS     R16, OP1         ;carrega R16 com valor do OP1
   LDS     R17, OP2         ;carrega R17 com valor do OP2
   ADD     R16, R17         ;soma e guarda resultado em R16
   STS     RES, R16         ;devolve res para memoria de dados
```

Fonte: elaborada pelo autor.

A primeira instrução é a **LDS R16, OP1**, a primeira linha abaixo do **.ORG** 0, o qual indica o início do programa. Como consequência dessa instrução, o valor contido no endereço 0x109 da memória de dados, correspondente ao operando 1, é transferido para dentro do núcleo, no registrador de trabalho R16. Perceba que não é o valor 0x0109 que é transferido para o R16, mas sim o valor que está na memória de dados, no endereço 0x0109. Isso é definido pela instrução LDS, que considera OP1 como um endereço, e não um valor.

Para transferir a constante 0x0109 para o R16, a instrução deveria ser **LDI R16, OP1**. Na segunda instrução ocorre a mesma ação para o segundo operador, trazido do endereço seguinte para o registrador de trabalho seguinte. Com os valores a serem processados dentro do núcleo, na terceira instrução os valores são somados na ULA e o resultado da operação é devolvido para o registrador geral R16. Na última instrução, o resultado é devolvido do núcleo para a memória de dados, no endereço 0x10B. As operações de transferência interna de dados, como a LDS e a STS não causam alterações nas flags do registrador de status. No entanto, como foi visto na Seção 1.2 desta unidade, no registrador de status, as instruções aritméticas podem acionar os bits ZERO, SIGN, OVERFLOW, NEGATIVE e CARRY, de acordo com o resultado da operação.

Avançando na prática

Controle do nível de um reservatório automático

Descrição da situação-problema

Devido à grande demanda que existe no seu Estado, você decidiu, depois de formado, montar uma empresa de automação

agrícola e já no seu primeiro projeto lhe foi solicitado, por um comerciante local, que automatizasse o sistema de abastecimento das caixas d'água e reservatórios, pois estes ainda eram abastecidos manualmente. Você deve, portanto, como projetista, entregar um laudo técnico descritivo (composto pelo fluxograma, algoritmo e programa em *Assembly*, correspondente, que deve ser embarcado no ATmega328) que permita a automação do sistema de nível dos reservatórios. Por se tratar de um sistema muito simples, você decidiu que faria um programa em *Assembly*, embarcado em uma placa do Arduino para controlar a bomba da caixa d'água.

A partir do exposto, responda: qual número mínimo de sensores de nível deve ser usado, para que o sistema não permita que a bomba seja acionada por curtos intervalos de tempo, ou em forma de surtos (chaveamentos liga/desliga sequências)? Considerando que o sensor e a bomba devem estar conectados ao sistema pelas portas digitais, construa o fluxograma e o algoritmo que o processo deve seguir, bem como o programa em *Assembly* correspondente, que deve ser embarcado no ATmega328.

Resolução da situação-problema

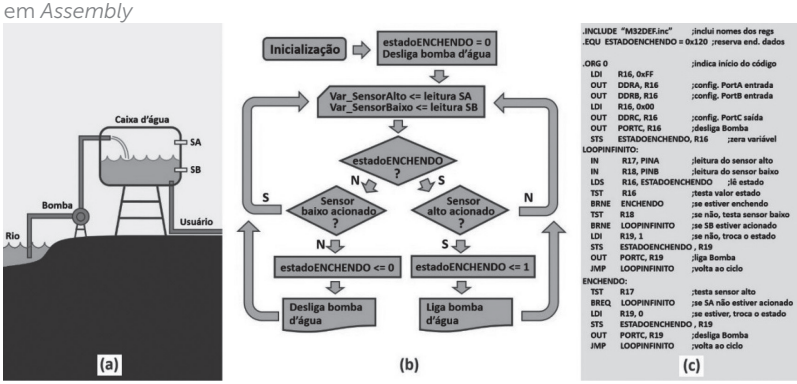
Você pode perceber que o uso de dois sensores de nível seria ideal para automatizar o sistema, um na parte superior (sensor alto), e outro na parte inferior (sensor baixo) do reservatório. Aparentemente um sensor alto bastaria para orientar o sistema, onde, de forma combinacional, a bomba está desligada caso o sensor acuse contato com a água, e ligada caso contrário.

Considerando que o fluxo para o exterior não é controlado pelo sistema, pois ocorre de acordo com o uso da água, quando a água atingir o nível do sensor, a bomba seria desligada, mas logo que alguém usasse um pouco de água, o nível desceria e a bomba seria religada, e por pouco tempo, pois o nível de água estaria próximo do sensor. Além do mais, caso o nível d'água esteja poucos milímetros abaixo da detecção do sensor, qualquer distúrbio no espelho d'agua, como oscilações, faria a bomba ser ligada e desligada sequencialmente, podendo danificá-la. Nesse tipo de situação, é ideal a criação de um sistema sequencial através de uma máquina de estados. Desse modo, quando o processo está no estado ENCHENDO = 1 (bomba ligada), ele deve permanecer nesse estado até que a água toque no sensor alto, fazendo o

sistema desligar a bomba e migrar para o estado ENCHENDO = 0. Nesse estado, apenas o sensor baixo é monitorado e o nível de água vai descendo conforme o seu uso. Quando o SB não detecta mais água, significa que a reservatório está vazio e a bomba e o estado devem ser acionados novamente. Esse simples mecanismo evita completamente os problemas que devem ocorrer com o uso de um único sensor, mencionados anteriormente.

Considerando que o sensor alto será conectado no sistema pelo periférico PORTA, o sensor baixo pelo PORTB e a bomba será acionada pela PORTC, um possível fluxograma e código *Assembly* para resolver este problema podem ser vistos na Figura 1.20:

Figura 1.20 | (a) Representação física do problema; (b) fluxograma e (c) programa em *Assembly*



Fonte: elaborada pelo autor.

Faça valer a pena

1. Existem muitas formas de criar um programa computacional, o que pode ser feito através de várias linguagens de programação. Algumas delas foram criadas para fins específicos, como desenvolvimento de jogos, e outras serviram apenas de base para a construção de linguagens mais avançadas. Apesar de algumas dessas linguagens não terem prosperado como se imaginava, muitas delas, modernas ou não, se tornaram um padrão mundial e são usadas atualmente pelos desenvolvedores de tecnologia da informação, como o Java.

Em relação à construção de programas embarcados em linguagens de máquinas, é correto afirmar:

a) Que se perpetuou no mercado e é usada até hoje com linguagem predominante na construção de sistemas embarcados modernos.

b) Que é aplicada exclusivamente para protocolos de troca de informações entre computadores.
c) Que é usada na criação de sistemas em rede, que devem empregar essa linguagem nas mensagens trocadas.
d) Que era usada nos primeiros códigos mas atualmente está em desuso, uma vez que linguagens de mais alto nível permitem um desenvolvimento muito mais rápido e eficiente.
e) Que é hoje usada apenas para a construção de máquinas de calcular, pois esta linguagem permite um melhor aproveitamento dos recursos, como display mostrador e botões numéricos.

2. Todos os programas de computador escritos carregam consigo um algoritmo correspondente, que pode ser representado em um fluxograma. O que acontece na prática é que alguns programadores constroem o fluxograma antes do código, já outros preferem programar diretamente. Esse fluxograma determina o comportamento do processo, de acordo com a ocorrência dos eventos tratados. Existem vários padrões de fluxogramas e modelos de algoritmos já estabelecidos, que se aplicam para as mais diversas situações e finalidades.
Pensando exclusivamente em programação para sistemas embarcados, considere as seguintes afirmações sobre o fluxo do código:

I – O sistema inicia-se em um *loop infinito*, onde permanece até que hardware seja todo ajustado e, em seguida, entra em uma fase de "finalização", onde executa os comandos necessários, encerrando assim o seu trabalho.
II – O programa principal se mantém sempre atualizando os valores das saídas, independentemente dos estados das entradas, usadas apenas para processamentos internos.
III – Antes de o sistema entrar no seu ciclo de trabalho, em um *loop infinito*, este deve ser inicializado, o que ocorre apenas uma vez e serve para configurar o conjunto para operar propriamente.
IV – Os desvios condicionais do fluxo do programa não podem ser usados em processos que controlam apenas uma saída, pois esta deve ser tratada na função principal do código.

Considerando as afirmativas, assinale a alternativa que apresenta a sequência correta:
a) F, V, V, V.
b) F, F, V, V.
c) V, F, F, F.
d) F, V, V, F.
e) F, F, V, F.

3. O microcontrolador ATmega328 atua sobre o conjunto de instruções da AVR. Essas instruções estão classificadas basicamente em quatro grandes grupos: instruções aritméticas/lógicas, instruções de transferência de dados, instruções de desvio de programa e instruções de controle interno do microcontrolador.
Os caminhos percorridos pelo programa em seu fluxograma são feitos pelas instruções de desvio, que podem ser incondicionais ou condicionadas a algum resultado usado como critério de decisão. Cada uma delas tem um propósito específico e deve ser empregada no contexto apropriado para surtir o efeito desejado.
Em relação à instrução em *Assembly* para o AVR de mnemônico BREQ, está correto afirmar que:

a) Processar o último dado que foi transferido para o núcleo, indicado pelo rótulo com a instrução.
b) Desviar o programa para onde o seu rótulo aponta, caso o resultado da última operação seja zero.
c) Transferir o dado que está onde o seu rótulo aponta, para que este possa ser processado no núcleo.
d) Desviar o programa para onde seu rótulo indica, independentemente de qualquer resultado anterior.
e) Invocar a sub-rotina, indicada pelo seu rótulo, independentemente de qualquer resultado anterior.

Referências

ATMEL AVR. **ATmega328/P *Datasheet Complete***. 2016. Disponível em: <http://www.atmel.com/pt/br/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso em: 29 abr. 2017.

BAER, J. L. **Arquitetura de microprocessadores.** 1. ed. Rio de Janeiro: LTC, 2013.

CERUZZI, P. E. **A history of modern computing.** 2. ed. Massachusetts: The MIT press, 2003.

D'AMORE, R. **Descrição e síntese de circuitos digitais**. 2. ed. São Paulo: LTC, 2012.

GRIER, D. A. **When computers were women.** New Jersey: Princeton University Press, 2005.

IFRAH, G. **The universal history of computing:** from the abacus to the quantum computer. New Jersey: John Wiley and Son, 2001.

LIMA, C. B. D.; VILLAÇA, M. V. M. **AVR e Arduino:** técnicas de projeto. Florianópolis, 2012.

MARGUSH, T. S. **Some Assembly required:** Assembly language programming with the AVR microcontrollers. 1. ed. New York: CRC Press, 2011.

MIZRAHI, V. V. **Treinamento em linguagem C.** New Jersey: Pearson, 2008.

REGAN, G. O. **A brief history of computing.** 2. ed. New York: Springer, 2012.

STALLINGS, W. **Arquitetura e organização de computadores.** 8. ed. São Paulo: Prentice Hall, 2009.

Programação embarcada

Convite ao estudo

Na Unidade 1, estudamos os principais conceitos sobre os sistemas de comutação, nesta segunda unidade, estudaremos uma ferramenta fundamental no desenvolvimento de produtos eletrônicos processados: a programação em linguagem C, com ênfase em sistemas embarcados. Para todos os componentes de um programa em C existe uma ou mais instruções equivalentes em linguagem *Assembly* (ou de máquina), e no início desse estudo compreenderemos a correspondência entre um código em C e o seu código embarcado (linguagem de máquina), ou seja, compilado em zeros e uns. Em seguida, relembraremos todos os elementos básicos para a construção de um programa em C. "Relembrar" porque é esperado que você já tenha estudado alguma linguagem de programação de mais alto nível sobre o *Assembly*, como Java, C, FORTRAN, BASIC, entre outros presentes nos cursos de Engenharia e TI (Tecnologia da Informação). Assim, esse estudo será uma grande revisão de conceitos e aplicações da linguagem C, direcionados para desenvolvimento de sistemas embarcados. Se você já possui grande experiência com a linguagem C, encontrará mais facilidade nesta unidade. No entanto, não salte nenhum tópico, o conteúdo aqui é indispensável para sua bagagem. Aproveite para absorver todos os detalhes que não estavam claros, além das particularidades dos sistemas embarcados, como o sincronismo de tarefas em tempo real. Por outro lado, se você não possui experiência sobre programação, ou já estudou, mas de forma superficial, não se preocupe. Toda a informação básica necessária estará nesse material, com uma sequência e detalhamento pensados para tornar seu estudo completo e eficaz. Ao fim das seções desta unidade, depois de estudar os

principais aspectos da linguagem C para microcontroladores, aplicaremos a seguinte situação-problema:

Uma empresa de automação residencial consolidada no mercado e preocupada em trabalhar com as tecnologias mais modernas, sempre atualizando os seus projetos de produtos e serviços, contrata você como o responsável pelos projetos de eletrônica e automação. Apesar de ainda não ter experiência com periféricos, você, ao fim desta unidade, estará pronto para elaborar algoritmos e implementá-los na linguagem C, para as situações mais simples.

Vamos começar?

Seção 2.1

Revisão de algoritmos e introdução à linguagem C

Diálogo aberto

Agora que você está ficando mais experiente com os sistemas embarcados, e já sabe elaborar algoritmos básicos e implementar em um código em *Assembly*, será apresentado a um método de linguagem e algoritmos mais sofisticados. A linguagem C, com a sua sucessora, C++, é a mais utilizada atualmente para o desenvolvimento de sistemas embarcados, pois é completa para atender às necessidades de desenvolvimento, é abstrata, porém, permite acesso direto à memória, aos registradores e ao núcleo, e foi elaborada para isso. Vamos estudar os elementos usados na programação em C, como implementar algoritmos através dos comandos de controle, aplicados para situações mais complexas, como sincronismo de tarefas paralelas em tempo real. Como visto anteriormente, você deve gerar uma solução para o problema abordado nesta seção. Fique tranquilo, pois aqui estão todas as informações que você precisa para resolver esse problema sem grandes dificuldades. Apenas prossiga em cada parte depois de ter realmente entendido o que leu. Nessa situação-problema, você foi contratado por uma renomada empresa de automação residencial. Inicialmente, você será o profissional da área de eletrônica e automação, responsável pela criação de um novo projeto: um sistema de automação de um portão de garagem, a partir de um controle remoto.

Como qualquer projeto, inicialmente são feitos vários protótipos, onde os primeiros são básicos, e os seguintes vão recebendo novas funcionalidades e adaptações, o que permite uma melhor depuração dos problemas que sempre aparecem. Dessa forma, essa primeira proposta será mais simples, apenas para gerar alguns resultados úteis no processo. Considerando que o chip usado para a construção do equipamento será um ATmega328, a sua atividade agora é elaborar um fluxograma, e a partir dele um programa básico em C, que atue e controle o portão de forma eficiente, a partir das três entradas: botão de acionamento (abre e fecha) - BOT, sensores fim de curso alto - SA, e baixo - SB (detecta portão fechado). O portão pode ser acionado

pelo sistema através de duas saídas: um relé para subir - RS, e outro para descer - RD, que claramente nunca podem ser acionados ao mesmo tempo, pois danificaria o motor. Considere estes sinais de entrada e saída como variáveis binárias, assim, um teste para saber o estado do botão pode ser feito: "se BOT acionado faça", "if (BOT == 1) " ou "if (BOT) ". O acionamento do relé para subir pode ser executado: "aciona RS" ou "RS = 1". Lembre-se de que você pode utilizar variáveis de estado ou contadores. Uma das questões que devem ser decididas antes de começar um projeto embarcado é se o circuito digital será de lógica combinacional ou lógica sequencial, para então implementar o programa. Sabendo que, logo depois de acionado, o portão não toca nenhum sensor fim de curso, até que chegue no destino final, justifique no seu relatório se o sistema que você criará deve possuir lógica sequencial ou combinacional, e quais são as condições que definem essa escolha. Informe também se o algoritmo do projeto deve ou não ser construído obrigatoriamente a partir de uma máquina de estados. Por fim, descreva no registro interno de desenvolvimento desse novo modelo, como a lógica do algoritmo pode ser implementada em um programa. Se o único método conhecido é por "máquina de estados", ou se existe(m) outra(s) maneira(s), e qual(is) é(são). Isso servirá como parâmetro de análise para você e outros projetistas decidirem as estruturas dos próximos algoritmos que serão feitos.

Temos trabalho pela frente, bons estudos!

Não pode faltar

Agora que você já passou por ela, deve ter percebido o quão importante foi o estudo da primeira parte, que são os fundamentos de sistemas embarcados. Vamos então entender como os programas são escritos em linguagens de mais alto nível e como as partes básicas estudadas se comportam durante a execução desses programas.

Vale ressaltar a importância de conhecer esses conceitos básicos para se tornar um bom profissional de eletrônica embarcada, pois no processo de construção de sistemas mais complexos, é comum que apareçam problemas ou comportamento inesperados, e se gasta muito tempo em processos de debug, ou depuração. O profissional que conhece todas as camadas está muito mais bem preparado para

resolver estas questões, e muitas vezes o problema se encontra nas camadas inferiores, as que fazem interface com o hardware.

Código *Assembly* equivalente

A linguagem C permite que trechos de códigos mais longos, complexos e trabalhosos de serem construídos em linguagem *Assembly* sejam feitos de forma mais sintética e simples. Essas qualidades tornam o processo de programação muito mais eficiente, o que ficará claro ao longo desta seção. Vale ressaltar que o tipo de conjunto de instruções aplicado para os chips AVR é o RISC (*Reduced Instruction Set Computer*, ou computador com conjunto reduzido de instruções (MARGUSH, 2011).

Pesquise mais

Veja as principais diferenças entre os dois tipos de conjunto de instruções (também conhecidos como tipos de arquitetura ou de máquina) no link a seguir. Disponível em: <http://www.sistemasembarcados.org/2015/11/15/processadores-arquitetura-risc-e-cisc/>. Acesso em: 26 jun. 2017.

Instruções de transferência de dados

Os comandos para este tipo de ação em C são bem simples, e todos eles utilizam o sinal de igualdade para transferir o elemento que está à direita para o elemento que está à esquerda da igualdade. Dessa forma, esse sinal ganha um novo significado na programação, que pode ser chamado de "recebe", ou seja, a expressão X=X+1, incoerente matematicamente, deve ser interpretada na programação como "X recebe o valor anterior de X mais um". Uma outra característica importante da linguagem C é que não precisamos trazer os valores para os registradores gerais antes de processá-los, nem mesmo devolver os valores dos resultados para a memória, uma vez que isso é gerado de forma automática pelo compilador. Apesar de ser permitido, não é uma prática em C escolher os endereços de memória para as variáveis, pois isso também é responsabilidade do compilador.

Exemplificando

Podemos escrever em C, portanto, as instruções de movimento de dados LDI, LDS, STS, IN, OUT e MOV, através do simples uso do símbolo igual. Podemos ver um exemplo na Figura 2.1:

Figura 2.1 | Instruções em *Assembly* e comandos em C equivalentes

Linguagem *Assembly*

```
.INCLUDE  "M32DEF.inc"  ;inclui nomes dos regs
.EQU  OP1VAR = 0x109    ;reserva end. dados para variável
.EQU  OP2VAR = 0x10A
.EQU  RESVAR = 0x10B

.ORG 0                  ;indica início do código
   LDI    R16, 5        ;carrega variáveis
   STS    OP1VAR, R16
   LDI    R16, 12
   STS    OP2VAR, R16
     •••
     •••
   LDS    R16, OP1      ;carrega R16 com valor do OP1
   LDS    R17, OP2      ;carrega R17 com valor do OP2
   ADD    R16, R17      ;soma e guarda resultado em R16
   STS    RESVAR, R16   ;devolve res para mem. de dados
     •••
     •••
```

Linguagem C

```
#include <avr/io.h>                 //inclui nomes dc

int  OP1VAR, OP2VAR, RESVAR;        //declara variáveis

Main{                               //indica início do
  OP1VAR = 5;                       //carrega variáve
  OP2VAR = 12;
     •••
     •••
  RESVAR = OP1VAR + OP2VAR;  //realiza a soma
     •••
     •••
```

Fonte: elaborada pelo autor.

Instruções lógicas e aritméticas

São todas feitas através dos operadores lógicos e aritméticos, diretamente escritos na forma convencional. Os operadores aritméticos usados são: +, - * e /, e para as operações lógicas: & (AND ou E), | (OR ou OU), ~ (NOT ou NEGAÇÃO/INVERSÃO) e ^ (XOR ou OU EXCLUSIVO). Lembramos que essas instruções são feitas bit a bit, e não considerando a variável como um único elemento, ou seja, 0x5A & 0xF0 → 0x50. Também podem ser usados parênteses para representar expressões, ou múltiplas operações. Dessa forma, se as operações envolverem apenas constantes, esse cálculo é feito pelo compilador na hora da compilação, inserindo no programa a ser embarcado apenas o resultado final. Por exemplo: VAR1 = (0x5A | 0xB2) & 0x0F. Vale relembrar que as operações dentro dos parênteses são feitas primeiro. Mais uma das vantagens da linguagem C é a compactação de operações lógicas e aritméticas, como i++; ao invés de i = i +1:, e x &= 0x01;, ao invés de x = x & 0x01;. Para deslocamento (não circular) de bits, usamos >> ou <<. Ex.: x = 0xA2 >> 4; (x←0x0A).

No entanto, quando as operações dependem de alguma variável, ou seja, um valor que não pode ser previsto na criação do programa, pois é dado a partir das condições do sistema em que estarão sendo

executadas, as operações lógicas e aritméticas são convertidas automaticamente nas instruções em *Assembly* equivalentes pelo compilador. Ex.: VAR1= (0x42 | VAR2) & VAR3.

Instruções de salto

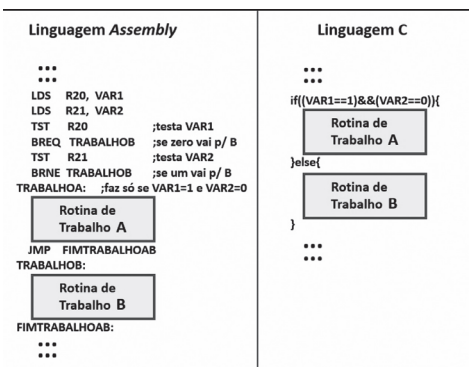
Essas instruções em *Assembly* não estão diretamente mapeadas em comando em C, mas existem alguns padrões construídos a partir delas que correspondem aos comandos de controle presentes em C, que serão estudados mais adiante, ainda nesta seção: IF, IF/ELSE, FOR, WHILE, DO WHILE, SWITCH, GOTO, BREAK e CONTINUE.

Lembrando que existem dois tipos de instrução de salto, condicionadas e incondicionais. Podemos observar que as instruções de salto condicionadas correspondem ao início dos comandos de controle, onde a condição é testada para saber se o bloco correspondente deve ser executado ou não. Já as instruções de salto incondicionais estão associadas aos términos dos comandos de controle, representados pelo "fecha chaves" (}), que indicam o fim do bloco.

Exemplificando

Podemos observar também que, quando a condição para executar um bloco ou trecho de programa não é singular, ou seja, múltiplas condições devem ser observadas para o tratamento, estas são feitas em *Assembly* de forma sequencial, como pode ser visto no exemplo da Figura 2.2.

Figura 2.2 | Saltos em *Assembly* e comandos de controle em C com condições



Fonte: elaborada pelo autor.

A partir dessa estrutura em linguagem *Assembly*, correspondente ao comando de controle IF, em linguagem C, podemos perceber que os comandos em laços (loops) em C podem ser convertidos para o *Assembly* na mesma forma, apenas com a adição de um salto incondicional ao fim do trecho, com retorno para a condição inicial, necessária para o programa "entrar" no laço (MAZIDI; NAIMI; NAIMI, 2010).

Diretivas de compilação

As diretivas em C, que são indicadas pelo prefixo "#", possuem o mesmo papel que as estudadas para o *Assembly*, com algumas pequenas modificações:

#include - corresponde identicamente ao ".INCLUDE" usado para incluir outros arquivos a serem considerados pelo compilador. Essa diretiva é muito usada quando se deseja criar programas em múltiplos arquivos, ou programas modularizados, subdivididos em funções, o que será estudado por nós na próxima seção desta unidade. Como feito anteriormente, haverá, em todos os programas em C criados por nós, o comando #include <avr/io.h> na primeira linha, que é o "cabeçalho AVR padrão", ou seja, contém todos os nomes dos registradores para usarmos na programação, ao invés de seus endereços.

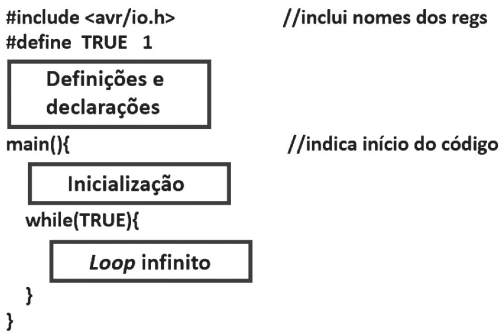
#define - essa diretiva foi mencionada anteriormente e é muito utilizada por várias linguagens de programação. Pode ser usada para definir constantes ou macros, como era feito com a diretiva de *Assembly* ".EQU", que além disso também era usada para definir variáveis. Em C, as variáveis são simplesmente declaradas antes de serem usadas, e possuem um escopo, que corresponde à região do programa em que pode ser visualizada/acessada, estudados logo adiante.

A diretiva em *Assembly* usada anteriormente para estipular o início do código ".ORG 0", não é representada por uma outra diretiva em linguagem C, mas pela função "main", que significa principal. Dessa forma, foi estipulado para a linguagem C, que todas as funções principais devem possuir o mesmo nome, main, (onde o programa se inicia) diferente de algumas outras linguagens. Esta deve estar sempre presente, mesmo que exista outras funções.

Estrutura do programa

Como foi estudado na seção anterior, todo o programa embarcado segue o padrão básico de, quando ligado, executar a parte da inicialização, onde o sistema é configurado, e ocorre uma única vez, e em seguida entrar no seu ciclo de trabalho, onde permanece até ser reiniciado. Dessa forma, com o uso de algum comando de iterações, devemos criar um loop infinito, ou seja, condicionado a uma exigência que será sempre satisfeita, como pode ser visto na Figura 2.3.

Figura 2.3 | Estrutura básica de um programa em C para sistemas embarcados com AVR



Fonte: elaborada pelo autor.

Reflita

Se para um determinado produto ou projeto, a parte de inicialização será sempre igual (ela é necessária) todas as vezes que este for iniciado, por que os periféricos devem ser configurados novamente sempre que ligados?

Comandos de controle

Apesar de alguns terem sidos mencionados, vamos esclarecer aqui o que são e como são usados esses comandos em linguagem C, responsáveis por ditar o fluxo do programa, equivalentes às instruções de salto condicionais em *Assembly*. As condições lógicas devem ser construídas com os operadores >, >=, <, <=, == (igual), != (diferente), ! (inverte), && e ||., onde estes dois últimos são as operações lógicas AND e OR, respectivamente, mas não no modo bit a bit, como vimos anteriormente, mas considerando a variável

como um único elemento, ou seja, o resultado da operação VAR1 **&&** 0x01 será um valor booleano, verdadeiro ou falso, diferentemente da operação VAR1 **&** 0x01, que resulta um vetor de bits. Note a diferença entre == e = (recebe, e não, testa).

IF - significa a condição "se", no modo: "se a condição for verdadeira, execute isto" onde isto se refere ao trecho de programa que está delimitado pelas chaves do respectivo IF. Por exemplo:

IF(VarPositiva==0){ Comando1; Comando2; } (BARNETT; COX; O'CULL, 2007).

Devemos ressaltar que para as decisões lógicas, a linguagem C considera qualquer valor diferente de zero como verdadeiro (inclusive negativos), e o valor zero como falso. Dessa forma, o comando em C acima pode ser reescrito na forma IF(!VarPositiva){ Comando1; Comando2; }.

Pois os comandos também serão executados apenas se a VarPositiva for nula, uma vez que zero é falso, e este está sendo invertido pelo operador !, tornando a condição verdadeira.

Assimile

Apesar de não ser usual nem recomendado, um programa em C pode ser escrito inteiro em uma linha, pois para o compilador isso não faz diferença, os comandos são limitados pelo símbolo ";". A única coisa que é levada em conta para distinção de linhas são as diretivas de compilação e os comentários, que são tudo que vier depois de "//", naquela linha. Podemos usar também o modo /***comentário** */, que pode ser quebrado em linhas.

O comando de controle IF pode ou não ser acompanhado do ELSE. Quando o programador não deseja que nada diferenciado seja feito quando a condição não for verdadeira, o ELSE torna-se desnecessário. Entretanto, quando o tratamento de uma condição requer dois casos diferenciados, o ELSE deve ser usado. Exemplo: **IF(Var1==4){ comando1;}ELSE{ comando2;}.**

Existe uma forma diferente para usar este comando, que é usado principalmente quando se deve escolher entre dois valores a serem carregados para uma variável, e a escolha depende de uma condição lógica. É através do operador ternário: "?" com ":". Apesar do "se" estar implícito, não se vê a palavra IF nesse caso. Assim, **IF(Var1==4)**

{ Var2 = 0; }ELSE{ Var2 = 1;} pode ser reescrito na forma **Var2 = (Var1==4)? 0:1;**. Apesar de incomum, esse operador também pode ser usado genericamente. Por exemplo **IF(Var1==4){cmd1; cmd2;} ELSE{ cmd3; cmd4;}** pode ser substituído por **(Var1==4)? (cmd1, cmd2) : (cmd3, cmd4);**. Lembrando que não é necessário o uso de parênteses na condição dessa operação, mas ajuda na visualização. Existe uma outra particularidade do IF que é poder não usar chaves, mas, apenas se existir um único comando (ou função) dentro do comando IF. Vale o mesmo para o ELSE, independentemente.

FOR - serve para criar laços (loops), geralmente com um número específico de repetições. Sua sintaxe é composta por três campos. O primeiro corresponde aos comandos de inicialização, ou seja, tudo que deve ser feito para preparar o programa para iniciar os loops. Esse campo é executado apenas uma vez, antes de iniciar o primeiro ciclo, não sendo repetido em cada ciclo. O segundo campo corresponde à condição lógica, que será verificada a cada iteração do laço, que será executada se a condição for satisfeita, ou seja, verdadeira. O programa se mantém realizando os loops enquanto a condição for verdadeira, e para quando a condição se torna falsa, continuando seu fluxo pelos comandos seguintes ao laço FOR. O terceiro campo contém a instrução que deve ser executada ao fim do ciclo, e geralmente é o incremento da variável usada para contabilizar o número de ciclos. Esses campos são separados pelo símbolo ";" e, apesar da condição para execução do ciclo ser um único bit, verdadeiro ou falso, presente no segundo campo, os campos primeiro e terceiro podem ser independentemente nulos ou múltiplos, tendo suas ações separadas por vírgula simples. Podemos perceber também que, tanto os comandos de inicialização dos laços, quanto os comandos feitos ao fim de cada loop, podem estar fora dos campos do FOR que surtirão o mesmo efeito, mas serve para melhor legibilidade. Assim, se uma rotina de trabalho A for feita 10 vezes seguidas, se usa **FOR(int i=0; i<10; i = i+1){ RotinaA}.**

Assimile

A rotina de trabalho corresponde a um trecho de programa, com as instruções/comandos escolhidos em sequência pelo programador, responsáveis por tratar/controlar/processar uma tarefa específica. Por exemplo, podemos fazer uma rotina para atualizar *8 leds* com o valor do resultado de uma operação, que pode ser genérica.

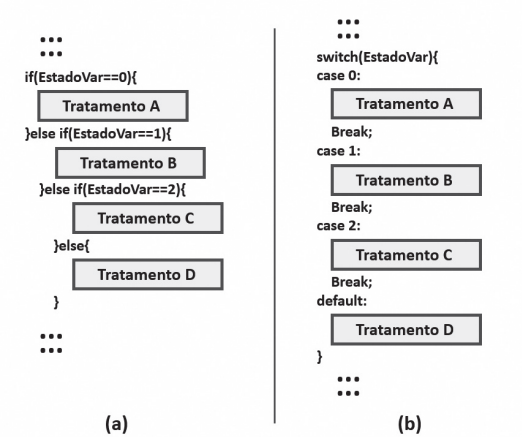
WHILE - significa "enquanto". Esse comando de controle possui apenas um parâmetro, ou seja, um campo a ser preenchido, que é a condição que deve ser satisfeita para que o laço seja executado. Geralmente, este comando é usado para executar um número de loops variável, que não pode ser previsto no ato da programação. É muito utilizado para criar o loop infinito do programa, que foi visto na Figura 1.12, na forma **WHILE(1){loop infinito}**, que também pode ser escrito como **FOR(; ;){loop infinito}**. O exemplo anterior do FOR também pode ser reescrito como: **int i=0; WHILE(i<10){ RotinaA; i++; }**. Existe também uma variação desse comando, que é utilizada quando se deseja executar o laço antes de testar: **DO{ RotinaA; i++; }WHILE(i<10);**.

SWITCH - significa "chaveamento". É útil quando se deseja criar um IF, porém, com mais de duas alternativas. Dessa forma, a variável testada não deve ser binária, mas possuir mais valores para associar um tratamento a cada um de seus valores esperados. Muito utilizada para se criar máquinas de estados, quando há mais de dois estados (MIZRAHI, 2008).

Exemplificando

Uma situação pode ser tratada de acordo com o estado da variável de estado "EstadoVar", que pode assumir de 0 até 3, visto na Figura 2.4:

Figura 2.4 | Uso do IF (a) e do SWITCH (b), em trechos equivalentes



Fonte: elaborada pelo autor.

BREAK - significa "parada". Podemos perceber o seu uso na Figura 2.4. Este comando é usado para interromper o trabalho em um laço de repetições, que podem ser WHILE, FOR e SWITCH. Quando o programa encontra esse comando, ele encerra as atividades daquele ciclo e continua seu fluxo com os comandos seguintes ao laço. Vale ressaltar que em caso de um **break** dentro de um loop, que, por sua vez, está dentro de outro loop (como um WHILE dentro de outro), este interromperá apenas um laço, o primeiro acima dele. Não podemos esquecer de colocar esse comando ao final de todas as opções do SWITCH, pois, caso contrário, todas as alternativas abaixo da selecionada serão executadas também (exceto se este efeito for desejado, é claro).

CONTINUE - significa "continua". Deve estar dentro de um laço, e quando o programa executa esse comando, as atividades naquele ciclo são encerradas, como no **break**. No entanto, ao invés do programa prosseguir com os comandos seguintes, ele volta a realizar o próximo loop normalmente. Isso torna apenas os ciclos em que foram executados o **continue** incompletos, sem a execução dos comandos que estão dentro do loop, e abaixo do **continue**.

GOTO - este comando desvia o programa para um rótulo que deve estar em qualquer parte do programa, seguido de dois pontos. Apesar da sua existência, não é aconselhável a sua utilização em programas em linguagem C, uma vez que ele realiza saltos incondicionais no software, tornando-o inelegível e de difícil manutenção. Um outro fator que confirma isso é o fato de não existir nenhuma situação em que o seu uso se faz necessário (KERNIGHAN; RITCHIE, 1991).

Lógica de circuitos digitais: combinacional x sequencial

Você provavelmente já ouviu falar desse tipo de classificação, quando estudou circuitos lógicos ou digitais. De fato, um sistema computacional será sempre sequencial, uma vez que utiliza memória, opera a partir de sinais de relógio, e executa as instruções sequencialmente. No entanto, podemos classificar as aplicações de maneira independente, o que é importante na análise de programas embarcados. De maneira direta, um circuito digital possui lógica combinacional se este não utiliza "memória" (em conceito, e não no chip), ou seja, não depende de eventos ocorridos no passado, e suas saídas podem ser diretamente mapeadas a partir das entradas

atuais. Também não pode haver realimentação, ou seja, saídas que dependem de alguma saída qualquer. Exemplo de aplicação embarcada: um led que acende quando um botão é pressionado. Caso a aplicação utilize algum valor do passado ("memória"), ou qualquer realimentação, o sistema será classificado como lógica sequencial, não permitindo a existência de sistemas "híbridos". Uma pergunta interessante é: antes do sistema existir, como definir qual será sua classificação? Quando todas as saídas podem ser diretamente mapeadas a partir das entradas, e esse mapeamento se mantém para todos os instantes do processo, o sistema é combinacional (uma tabela verdade sem estados ou realimentação). Caso contrário, é sequencial. Na automação de uma caixa d'água, por exemplo, quando o nível de água está no meio, entre os sensores alto e baixo (SB=1 e SA=0), o sistema não pode definir como será a saída da bomba, sem usar memória, pois pode estar nas fases "enchendo" ou "esvaziando". É necessário, então, implementar uma máquina de estados, ou usar realimentação, que é a mesma coisa, mas de maneira indireta.

Sincronismo de tarefas paralelas

Esse é um tema que abrange técnicas muito importantes para o desenvolvimento de projetos para aplicações de tempo real. Dentre estas, a mais utilizada, e que serve para todas as situações, é a utilização de variáveis como contadores, que a partir de uma base de tempo de contagem, também conhecido como *tick* (instante, momento), pode prover referência de tempo para alguma tarefa do programa. Essas também são conhecidas como técnicas de *timeout* (estouro de tempo). Dessa forma, se escolhe uma base de tempo que seja um divisor inteiro comum para todos os períodos utilizados pelas tarefas; se insere uma rotina de *delay* (atraso) no fim do loop infinito com essa base (*tick*); e se declara um contador para cada tarefa. Dessa forma, estas podem operar de maneira independente.

Exemplificando

Tarefa 1: Led1 (PortC) deve piscar em 5 Hz enquanto botão1 (PortA) estiver acionado, e ficar apagado caso contrário. Tarefa 2: O led2 (PortD) deve trocar seu estado, apenas uma vez, depois do botão2 (PortB)

permanecer pressionado por dois segundos seguidos (veja a lógica, abstraia o PortX).

Figura 2.5 | Sincronismo de duas tarefas independentes no ATmega328

```
#define F_CPU 16000000UL  //definição da frequência de clock para
                          //uso das funções da biblioteca delay

#include <avr/io.h>      //biblioteca para acesso aos registradores do uC
#include "util/delay.h“ //biblioteca para funções de delay

#define TRUE 1
#define LIGADO 1
#define DESLIGADO 0

#define PBOT1  PINA    //Porta ligada ao Botão 1
#define PBOT2  PINB    //Porta ligada ao Botão 2
#define PLED1  PORTC  //Porta ligada ao Led 1
#define PLED2  PORTD  //Porta ligada ao Led 2

int main(void){          //função principal

    unsigned int Cnt1, Cnt2, Bot1, Bot2;  //variáveis do programa

    DDRA = 0x00;          //configura porta A como entrada: Botão 1
    DDRB = 0x00;          //configura porta B como entrada: Botão 2
    DDRC = 0xFF;          //configura porta C como saída: Led1
    DDRD = 0xFF;          //configura porta D como saída: Led2
    PLED1 = DESLIGADO ;  //desliga Led1
    PLED2 = DESLIGADO ;  //desliga Led2

    Cnt1 = 0;             //zera contador 1
    Cnt2 = 0;             //zera contador 2

    while(TRUE){        //loop infinito

        Bot1 = PBOT1;      //leitura das entradas
        Bot2 = PBOT2;

        if(Bot1){                  //Tratamento da tarefa 1
            if(Cnt1<10) Cnt1++;
            else{                  //inverte estado do led1
                PLED1 ^= 1;        //a cada 10 x 10ms = 100ms
                Cnt1 = 0;          //->período de 200ms / F1=5Hz
            }
        }else PLED1 = DESLIGADO;      //desliga led1

        if(Bot2){                  //Tratamento da tarefa 2
            if(Cnt2<200) Cnt2++;
            else if(Cnt2==200){    //espera por 2 seg com o bot2
                PLED2 ^= 1;        //pressionado para inverter led2
                Cnt2++;            //(se Cnt2=201 não ocorre nada)
            }
        }else Cnt2 = 0;       //zera cont2 se soltar o bot2

        _delay_ms(10);          //aguarda 1/100 seg
    }
}
```

Fonte: elaborada pelo autor.

Sem medo de errar

Na situação-problema desta unidade, uma empresa de automação residencial consolidada no mercado contrata você como o desenvolvedor de projetos de eletrônica e automação. Você recebeu a tarefa de elaborar um algoritmo básico para o controle de um portão de garagem. Você deve, portanto, elaborar um fluxograma para este algoritmo, e, a partir deste, um programa em C. Este sistema automático deve controlar o portão de forma eficiente, a partir das três entradas: botão de acionamento (abre e fecha) - BOT, sensores fim de curso alto - SA, e baixo - SB (detecta portão fechado). Saídas: um relé para subir - RS, e outro para descer - RD. Você deve, por fim, fazer uma análise sobre a classificação da lógica do sistema, em combinacional, ou sequencial, e quais foram os critérios adotados, pois isso será usado nas próximas aplicações.

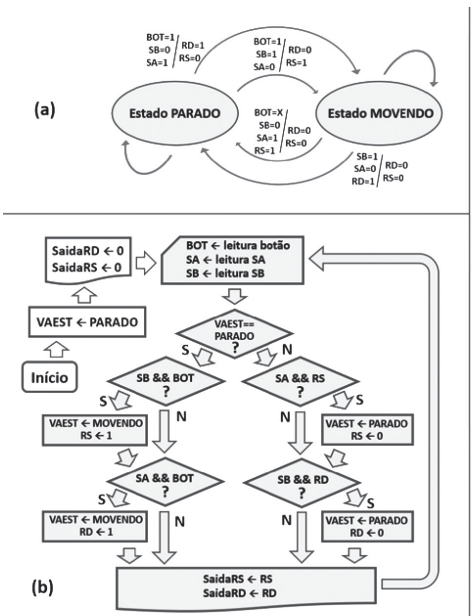
O começo é sempre mais complicado, mas uma boa forma de iniciar é imaginando o problema, de preferência com o rascunho de um desenho representando a situação. Deve-se anotar as variáveis, e imaginar o processo acontecendo, passo a passo, percebendo

como as variáveis devem ser atualizadas em consequência dos eventos esperados. Por exemplo, vamos criar uma máquina de estados binária, onde o sistema pode estar PARADO (zero) ou MOVENDO (um), de acordo com a variável de estados VAEST. Imaginamos assim o sistema em seu estado inicial, ou seja, o portão está fechado e parado, e a única variável não nula é o SB (detecta portão fechado). A única ação seguinte é a abertura do portão, que só é feita se o botão for acionado. Podemos perceber que aqui devemos "tratar" o controle do portão de acordo com uma condição, ou seja, o sistema deve, nessa condição (estado), acionar o RS caso o botão seja pressionado. Podemos observar também que é possível diferenciar se o portão está parado aberto ou parado fechado, testando os estados dos sensores. Dessa forma, o primeiro desvio do programa principal, que deve ocorrer para iniciar a abertura do portão, pode ser IF((VAEST==0)&&(SB==1) &&(BOT==1)){ VAEST=1; RS=1;}. Verifique que esta condição só pode ser satisfeita se o portão estiver parado fechado e o botão for pressionado, o que aciona o portão para descer e altera o estado do sistema para MOVENDO. Note que esta mesma condição pode ser simplificada para IF((!VAEST)&&(SB)&&(BOT)).

Agora que o portão está subindo (abrindo), devemos imaginar qual é a próxima atuação do sistema, que deve ser parar o portão quando estiver totalmente aberto, indicado pelo sensor SA. Assim, a condição a ser criada para isto pode ser IF((VAEST)&&(SA)){VAEST=0; RD=0}. Percebemos que agora o portão está parado e aberto, e nenhuma das duas condições anteriores são satisfeitas, como desejado. A próxima ação agora é fechar o portão quando o usuário pressionar novamente o botão, e este tratamento requer a seguinte condição lógica: IF((!VAEST)&&(SA)&&(BOT)){VAEST=1; RD=1;}, pois será satisfeita se o portão estiver parado aberto e o usuário acionar o botão, fazendo o portão fechar. A última ação a ser feita para fechar o ciclo é parar o portão quando estiver fechado, que pode ser realizado pelo tratamento: IF((VAEST)&&(SB)){VAEST=0; RD=0;}, o que desliga o portão e devolve o conjunto para a primeira etapa do processo. É importante salientar que essa não é a única maneira de se criar um algoritmo para este problema, mas uma versão simples e direta. Outra forma de se fazer é com uma máquina com 4 estados: PARABERTO, PARFECHADO, MOVSUB, MOVDESC, o que torna a programação ainda mais fácil. Uma terceira maneira seria

sem usar variáveis de estado, como pode ser visto na Figura 2.6. No entanto, apesar de neste exemplo os estados não aparecerem explicitamente, ainda existem e estão representados pelo uso das saídas nas condições de tratamento, como se pode observar na Figura, ou seja, se você verificar, as saídas e os valores são RD=1 e RS=0, já sabe que o estado é MOVDESC, sem precisar de uma variável para indicar. Sempre que o sistema depender de estados para controlar as saídas, que podem ser representadas por realimentação (saídas usadas como entradas, para tomar decisões), dizemos que o sistema é sequencial. Isso significa que, pelo menos em algum momento do processo, o sistema deve emitir saídas diferentes para um mesmo estado das entradas. Por exemplo, quando o portão está subindo, as entradas são todas nulas, e quando está descendo também, mas a saída não é igual nas duas situações. Quando as saídas podem apenas ser mapeadas a partir das entradas, ou seja, para cada condição das entradas existe um acionamento único das saídas, dizemos que o conjunto é combinacional, e isso depende da natureza do problema.

Figura 2.6 | Primeira solução proposta: com MDE: (a) diagrama da MDE; (b) fluxograma; (c) programa

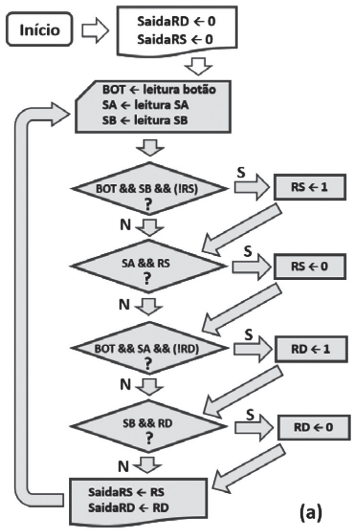



```
#include <avr/io.h>    //inclui nomes dos regs
#define  TRUE  1
#define ESTPARAD  0
#define ESTMOVENDO  1

main(){          //indica início do código
  unsigned int  SB, AS, BOT, RD, RS, VAEST;
  RD = 0;        //desliga relés
  RS = 0;
  VAEST = ESTPARAD;        //estado inicial
  while(TRUE){             //loop infinito
    SA = leituraEstadoSA;  //cmd fictício leitura
    SB = leituraEstadoSB;
    BOT = leituraEstadoBOT;
    if(VAEST== ESTPARAD){    //se estiver parado
      if(SB && BOT){         //se fechado e acionado
        VAEST = ESTMOVENDO;
        RS = 1;              //faz subir
      }
      if(SA && BOT){         //se aberto e acionado
        VAEST = ESTMOVENDO;
        RD = 1;              //faz descer
      }
    }else{                   //se estiver movendo
      if(SA){                //se tocou em SA
        VAEST = ESTMOVENDO;
        RS = 0;              //desliga motor
      }
      if(SB){                //se tocou sb
        VAEST = ESTMOVENDO;
        RD = 0;              //desliga motor
      }
    }
    RDSaida = RD; RSSaida = RS;  //cmd fictício saída
  }
}
```

(c)

Fonte: elaborada pelo autor.

Figura 2.7 | Segunda solução proposta: sem MDE: (a) fluxograma (b) programa




```
#include <avr/io.h>    //inclui nomes dos regs
#define  TRUE  1

main(){          //indica início do código
     unsigned int  SB, AS, BOT, RD, RS;
     RD = 0;       //desliga relés
     RS = 0;
     while(TRUE){              //loop infinito
          SA = leituraEstadoSA;  //cmd fictício leitura
          SB = leituraEstadoSB;
          BOT = leituraEstadoBOT;
          if(BOT && SB && (!RS)){  //se fechado e acionado
               RS = 1;                  //faz subir
          }
          if(SA && RS){         //se subindo e tocar em SA
               RS = 0;          //para
          }
          if(BOT && SA && (!RD)){       //se aberto e acionado
               RD = 1;                  //faz subir
          }
          if(SB && RD){         //se descendo e tocar em SB
               RD = 0;          //para
          }
          RDSaida = RD; RSSaida = RS;  //cmd fictício saída
     }
}
```

(b)

Fonte: elaborada pelo autor.

Se analisarmos com calma, podemos perceber que no código da primeira proposta existe um erro. Qual é esse erro? Como podemos consertá-lo? O que deve ocorrer se o sistema for executado com o programa nessa forma? (Dica: confira o diagrama e o fluxograma). Se não encontrar, pergunte ao seu professor.

Avançando na prática

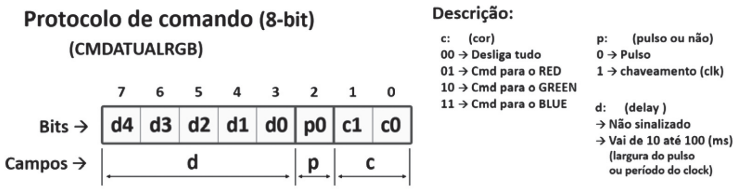
Controle automático de um misturador de tintas

Descrição da situação-problema

Você trabalha em uma empresa de automação e está no desenvolvimento do projeto de um misturador de tintas RGB - *Red Green e Blue* (Vermelho, Verde e Azul). Sabendo que o sistema será operado a partir de ATmega328, você recebeu a tarefa de elaborar um programa em C que controle as três saídas para as válvulas das tintas, independentemente. Quando o sistema está em funcionamento, cada uma dessas válvulas pode ser acionada de dois modos: através de um pulso de largura estabelecida, ou chaveamento contínuo, como o sinal de um clock (relógio), com frequência também estabelecida. O sistema que a sua equipe trabalha é escravo (subordinado) de um outro sistema de controle de mais alto nível, ou seja, o seu programa

não decidirá sobre os modos e os tempos de cada saída, apenas fará o acionamento e o controle a partir das informações trazidas do sistema "mestre". Considere as duas variáveis, compartilhadas entre os dois sistemas, CMDATUALRGB, e FLAGCMDNOVO, onde esta última indica se chegou um novo comando, e pode ser apagada pelo seu programa. A primeira traz o comando, que pode ser sobrescrito, e que segue as especificações descritas na Figura 2.8:

Figura 2.8 | Especificações do protocolo usado em CMDATUALRGB



Fonte: elaborada pelo autor.

Considere também uma saída para um alarme (lâmpada), que deve ser acionada (piscar em 1 Hz) se o sistema mestre ficar mais de um segundo sem enviar comandos, ou se o campo d estiver fora das especificações. O alarme só deve cessar se receber o comando "desliga tudo" (válvulas).

Resolução da situação-problema

Uma possível solução para o problema:

Figura 2.9 | Proposta para solução do problema

```
#define F_CPU 16000000UL  //def freq de clock para
                          //funções de delay

#include <avr/io.h>     //bib com regs do uC
#include "util/delay.h"//bib de delay

#define TRUE      1
#define COR_R     1
#define COR_G     2
#define COR_B     3

#define DESLIGADO 0
#define LIGADO 1

#define P_RED     PORTA  //Porta ligada à válvula RED
#define P_GREEN  PORTB  //Porta ligada à válvula GRREN
#define P_BLUE    PORTC  //Porta ligada à válvula BLUE
#define P_ALARM PORTD  //Porta ligada ao Alarme

int main(void){              //função principal

    unsigned int dR, dG, dB;   //variáveis do programa
    unsigned int pR=1, pG=1, pB=1, cor;
    unsigned int cntR=0, CntG=0, CntB=0, CntAlarme=0;

    DDRA = 0xFF;          //config. porta A saída: RED
    DDRB = 0xFF;          //config. porta B saída: GREEN
    DDRC = 0xFF;          //config. porta C saída: BLUE
    DDRD = 0xFF;          //config. porta D saída: Alarme
    P_RED = DESLIGADO ;       //desliga válvula RED
    P_GREEN = DESLIGADO ;     //desliga válvula GREEN
    P_BLUE= DESLIGADO ;       //desliga válvula BLUE

while(TRUE){      //loop infinito

    if(FLAGCMDNOVO){
        Cor = CMDATUALRGB & 0x03; //separa campo c
        if(!Cor){       // se for "DESLIGA TUDO"
            pR = 1; CntR = 0;
            pG = 1; CntG = 0;
            pB = 1; CntB = 0;
            P_ALARM = DESLIGADO;
        }else{          // se não, decodifica cmd
            switch(cor){
            case COR_R:    //checa se é para a valv. RED
                pR = (CMDATUALRGB >> 2) & 0x01;
                dR = (CMDATUALRGB >> 3) & 0x1F;
                if(pr){
                    CntR = dR;  //se for pulso já aciona
                    P_RED = LIGADO;
                }
                if((dR<10)||(dR>100)) CntAlarme=1000;
                break;
            case COR_G: ...  //idem para GREEN.
            defalut:   ... //idem para  BLUE.
            }
        }
        if((!Cor) || (CntAlarme  <= 1000)) CntAlarme = 0;
        FLAGCMDNOVO=0;
    }

    if(pR){                    //Tratamento d
        if(CntR) CntR--;
        else P_RED = DESLIGADO;  //pulso
    }else{
        if(CntR<(dR/2)) Cnt1++;    //chave
        else{
            PLED1 ^= 1;
            CntR = 0;
    }
    if(pG){ ... }  //Tratamento da tarefa 2 (i
    if(pB){ ... }  //Tratamento da tarefa 3 (i
    }
    if(cntAlarme<1000) CntAlarme ++;
    else if(CntAlarme<1500){          //T
            P_ALARM = LIGADO;
            CntAlarme++;
        }else if(CntAlarme<2000){
                P_ALARM = DESLIGADO;
                CntAlarme++;
            }else CntAlarme=1000;

    _delay_ms(1);     //aguarda 1/1000 seg
}                     //(desprezando o tem
```

Fonte: elaborada pelo autor.

Faça valer a pena

1. As diretivas de compilação são comandos que são usados pelo compilador, mas não comandos que são instruções de um programa, e ajudam o programador aumentando a legibilidade do código.

A respeito das diretivas de compilação em linguagem C, voltada para o microcontrolador ATmega328, qual das alternativas a seguir pode ser considerada verdadeira?

a) A diretiva "#define" é obrigatória para todos os programas embarcados e deve ser usada para definir todas as variáveis do programa.

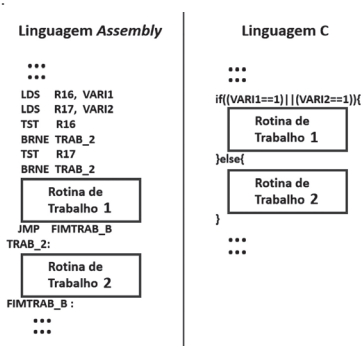
b) A diretiva "#include" é equivalente à diretiva de *Assembly* ".INCLUDE", que é utilizada para incluir algum arquivo para o seu programa. Apesar de não ser obrigatório, é muito útil na construção de programas embarcados, principalmente na primeira linha: #include <avr/io.h>, incluindo os nomes registradores para utilização do programador.

c) A diretiva "*main*(){" é utilizada para definir onde é o começo da memória de dados, ou seja, onde as variáveis do programa estarão para serem manipuladas. Todos os programas em linguagem C devem possuir pelo menos uma diretiva dessa, indicando a passagem para o loop infinito.

d) A diretiva ".EQU" é responsável por registrar um valor em uma variável, mas que pode ser alterada posteriormente pelo código através de outra ".EQU", diferentemente da diretiva "#define", que associa uma variável a um valor definitivamente.

e) A diretiva "*while*(TRUE)" serve para indicar onde estará no código o loop infinito, que pode ser unitário, mas pode ser múltiplo, se o programa demandar muitos processamentos. Essa diretiva não existe em *Assembly*, nem alguma correspondente.

2. A linguagem C permite que trechos de códigos mais longos, complexos e trabalhosos de se construir em linguagem *Assembly* sejam feitos de forma mais sintética, simples e abstrata, tornando o processo de programação muito mais eficiente.



Fonte: elaborada pelo autor.

A respeito da comparação entre eles, qual das alternativas a seguir é correta?
a) Não são equivalentes, mas seriam se a variável VARI1 fosse testada com o valor zero ao invés de um, no trecho em C (à direita), e se a operação lógica fosse XOR "^" (OU exclusivo), ao invés de OR "||"(OU).
b) São perfeitamente equivalentes, ou seja, vão produzir o mesmo efeito se forem processadas pelo ATmega328.
c) Não são correspondentes, mas seriam se a primeira instrução de salto condicionado em *Assembly* fosse "BREQ" ao invés de "BRNE", e se a variável VARI1 fosse testada com o valor zero ao invés de um no trecho em linguagem C.
d) Não se correspondem, e teriam que ser reescritas de outra maneira, pois não há nenhuma alteração que possa ser feita para torná-las equivalentes, devido às suas estruturas não permitirem isso.
e) Não são equivalentes, mas seriam se as duas variáveis fossem testadas com o valor zero ao invés de um, no trecho em C (à direita), e a operação lógica fosse AND "&&" (E), ao invés de OR "||" (OU), no mesmo trecho.

3. Um mesmo programa em linguagem C pode apresentar comportamentos diferentes cada vez que for executado. Isso se deve ao fato do programa não seguir sempre um mesmo caminho no seu fluxograma, que não pode ser previsto durante a programação, pois depende das condições que serão encontradas pelo sistema no momento em que for executado.
Todos os programas em linguagem C que são capazes de tomar decisões possuem comandos de controle em seu corpo. A respeito desses comandos, considere as seguintes afirmações:
I – O comando IF só pode ser usado com o ELSE e faz o programa percorrer o menor caminho, que exigirá menor processamento.
II – O comando BREAK deve estar sempre dentro de um laço, e serve para interromper a execução desse laço, fazendo o programa continuar seu fluxo com os comandos seguintes ao laço.
III – O comando SWITCH serve para criar laços quando não é possível prever quantas vezes o laço será executado, e permite que o programa se comporte de maneira diferente em cada vez que o laço for processado.
IV – O comando CONTINUE serve para fazer o programa continuar pelo seu fluxo normal, sem alteração ou salto. Deve ser inserido sempre ao fim do loop infinito para o programa não cessar sua execução, continuando com um ciclo seguinte.
Quais das afirmações podem ser consideradas verdadeiras ou falsas, respeitando a ordem?

a) V, F, V, F.
b) F, F, F, V.
c) F, V, F, F.
d) F, F, V, V.
e) F, V, V, F.

Seção 2.2

Programação em linguagem C

Diálogo aberto

Nesta seção, estudaremos os últimos aspectos da linguagem de programação C, com ênfase em sistemas embarcados e aplicações de tempo real. Para entender como esses elementos são devidamente reunidos para a construção de uma solução, vamos resolver algumas questões mais avançadas sobre o problema desta unidade. Aqui, estudaremos os tipos de dados mais avançados, como matrizes, estruturas e ponteiros, além das suas utilidades e aplicações. Vamos ver também como modularizar um problema complexo e pequenos problemas específicos através das funções. Apesar de ainda não ter experiência com periféricos, você, aluno, ao fim desta unidade estará pronto para elaborar algoritmos mais avançados e implementá-los na linguagem C, bem como resolver essa situação-problema. Nela, uma empresa de automação residencial consolidada no mercado e preocupada em trabalhar com as tecnologias mais modernas, sempre atualizando os seus projetos de produtos e serviços, contrata você como o responsável técnico de projetos de eletrônica e automação.

Assim que ingressou, você recebeu a responsabilidade de elaborar um algoritmo básico para o controle de um portão de garagem. Construa um fluxograma, e a partir dele um programa básico em C, que atue e controle o portão de forma eficiente, a partir das três entradas: botão de acionamento (abre e fecha) - BOT, sensores fim de curso alto - SA, e baixo - SB (detecta portão fechado). O portão pode ser acionado pelo sistema através de duas saídas: um relé para subir - RS, e outro para descer - RD, que claramente nunca podem ser acionados ao mesmo tempo, pois danificaria o motor. Considere estes sinais de entrada e saída como variáveis binárias, assim, um teste para saber o estado do botão pode ser feito: "se BOT acionado faça", "if (BOT == 1) " ou "if (BOT) ". O acionamento do relé para subir pode ser executado: "aciona RS" ou "RS = 1". Agora que já desenvolveu um algoritmo, você foi requisitado para escrevê-lo em linguagem C, além de inserir algumas adaptações para o sistema ficar mais robusto e preparado para a realidade. Os sinais continuam

sendo apenas variáveis, e agora você conta com mais dois sensores digitais, um detecta a presença do veículo abaixo do portão - SP, para evitar colisão, e o outro detecta sobrecarga (ou sobrecorrente) do motor - SS, indicando bloqueio mecânico. Também foi adicionada uma saída ligada a uma lanterna sinalizadora - LS, que deve piscar em 0,5 Hz quando o portão estiver em movimento ou aberto, e deve ser desligada 5 segundos após o fechamento do portão. O portão agora deve fechar de forma automática, caso o dono tenha esquecido, 10 segundos depois de permanecido aberto, sem risco de danificar o carro, é claro. Se durante o acionamento houver sobrecarga, o motor deve parar, a lanterna piscará por 5 segundos em 2 Hz, antes de desligar, e o portão deve voltar para sua posição anterior, até completar o curso e parar. Para ter referência de tempo, você deve usar a função genérica "delay_ms(X)" (que gera um atraso de X milissegundos na execução do programa). Não se esqueça que, durante o movimento, o sensor final não pode ficar longos períodos (> 50 ms) sem ser verificado, pois se este for acionado e isso não for detectado o quanto antes pelo sistema, o motor ficará ligado e a parte mecânica será danificada. Parece que essa é uma tarefa bem complicada, com muitas considerações, mas se você separar em partes, pensando em uma consideração por vez, verá que consegue facilmente criar todas em separado e depois unir em um único "loop infinito". Vamos lá!

Não pode faltar

Depois de estudar os comandos de controle, responsáveis pelas decisões sobre o fluxo do programa, nesta segunda seção, além dos tipos de variáveis, estudaremos os aspectos mais avançados da linguagem C.

Tipos de variáveis

Todas as informações manipuladas pelo programa devem pertencer a algum tipo, ou seja, como seu valor é representado digitalmente, e qual são os valores máximos e mínimos permitidos.

int - tipo inteiro padrão de tamanho oficial de 32-bit, apesar de permitir outros tamanhos, de acordo com a arquitetura da máquina que rodará o código. Para nós, o AVR considera esse tipo

com tamanho de 16-bit, que pode assumir valores negativos, assim o bit mais significativo representa o sinal e os demais a magnitude. Portanto, vai de -2^{15} até $2^{15}-1$. Pode ser escrita na forma mais completa, **signed int**, que é a mesma coisa.

unsigned int - tipo inteiro sem sinal, assume valores positivos de 16-bit. Vai de 0 até $2^{16}-1$.

char - tipo caractere, que assume valores de 8-bit inteiros sinalizados. Vai de -2^{7} até $2^{7}-1$.

unsigned char – igual ao char, mas se for tratado como número assume inteiros de 0 até $2^{8}-1$. Os tipos char são geralmente usados para tratar caracteres de texto, seguindo a tabela ASCII.

long - tipo inteiro sinalizado de 32-bit (ou o dobro do int), podendo ir de -2^{31} até $2^{31}-1$.

unsigned long - tipo inteiro positivo de 32-bit. Vai de zero até $2^{32}-1$.

float - tipo flutuante, assume valores fracionários ou não inteiros, sinalizados e que podem ter os valores de $-3.4*10^{38}$ a $-1.2*10^{-38}$, 0, e de $1.2*10^{-38}$ até $3.4*10^{38}$, precisão: 6 casas decimais.

double - também tipo ponto flutuante, mas com 64-bit de largura, vai de $\pm 2.3*10^{-308}$ a $\pm 1.7*10^{308}$, com precisão de 15 casas decimais.

Conversão de tipos: também chamado de *typecasting* (moldagem de tipo), serve para ajustar um tipo de variável em outro, quando precisamos processar variáveis diferentes juntas. Apesar de não ser obrigatório, pois este processo é feito automaticamente pelo compilador em caso de omissão, é extremamente recomendado o uso de *typecasting*, colocando entre parênteses o novo tipo desejado, quando o formato do dado precisar ser adaptado, como: **float Pi = 3.14152; unsigned int Res; Res = (unsigned int)(Pi / 0.5); //Res ← 6**

Pesquise mais

Veja quais são as utilidades dos qualificadores de tipo: const, static, extern e volatile. Disponível em: <http://www.pontov.com.br/site/cpp/61-aprendendo-o-c/242-parte-1-expressoes-em-c>. Acesso em: 20 jun. 2017.

Tipos complexos de dados

Já estudamos os tipos primários de variáveis para a programação em linguagem C, vamos agora estudar as matrizes, os ponteiros e as estruturas de dados.

Matrizes - são elementos de um mesmo tipo que estão amarrados pela matriz em que estão contidos. Podem ter quantas dimensões forem necessárias, apesar de geralmente se utilizar apenas uma, ou seja, variáveis como vetores. As dimensões e os tamanhos da variável são definidos no momento em que ela é declarada, sendo as dimensões indicadas pelo número de pares de colchetes seguidos ao nome da variável, e os tamanhos de cada dimensão estipulados pelo número dentro de cada par de colchete. Por exemplo: **float VarMatriz[4][2][6]**;. Essa matriz possui 3 dimensões, e 4, 2 e 6 representam a quantidade de elementos (*float*) em cada dimensão, respectivamente. Como o índice começa a ser contabilizado a partir do zero, o último elemento dessa matriz seria acessado como **VarMatriz[3][1][5] = 0.13142;**, por exemplo. Quando se declara uma matriz, deve-se definir o tipo dessa matriz, o que significa que todos os elementos matriz serão desse mesmo tipo, apesar de poderem carregar valores distintos. Dessa forma, se você precisar ter caracteres constantes representando as vogais para o seu código, deve usar no início dele a declaração **unsigned char Vetvogais[5]={'a', 'e', 'i', 'o', 'u'};**. Considerando que o primeiro elemento do vetor tem índice zero, se precisar transferir o caractere 'i' para a variável **unsigned char Vogalatual**, deve usar o comando **Vogalatual = Vetvogais[2];**. Para criar uma matriz de constantes, usa-se ";" para separar as linhas. Por exemplo, para se criar uma matriz, onde a primeira linha é composta pelos algarismos pares, e a segunda pelos ímpares, você deve declarar: **const unsigned int MatrizAlgs[2][5] = {0, 2, 4, 6, 8; 1, 3, 5, 7, 9};**. O número 7 pode ser acessado, posteriormente, por **MatrizAlgs[1][3]**. Vale notar, relembrando os primeiros conceitos estudados no início do nosso curso, que quando é usado o prefixo **const** para declarar constantes, esses valores são armazenados na memória de programa, e não na memória de dados, pois como o nome diz, são constantes e não precisam ocupar a memória volátil. Deve-se ressaltar também que, exclusivamente para a declaração de constantes, os valores dos tamanhos podem ser omitidos, uma vez que o compilador saberá seus tamanhos pela quantidade de elementos inseridos na matriz no momento da declaração. Por exemplo: **Vetvogais[]={'a', 'e', 'i',**

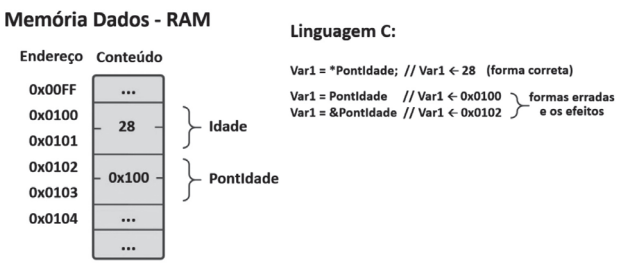
'o', 'u'};. O compilador saberá pela atribuição que a matriz Vetvogais possui 5 itens. Para variáveis, se o seu programa for processar, por exemplo, o peso (Kg) e a altura (m) de 10 pessoas, você pode usar a variável composta **float MatrizPesoAltura[2][10];** onde a primeira linha armazenará o peso da pessoa "1" até a pessoa "10", e a segunda linha a altura, igualmente. Assim, quando for armazenar o valor da altura da última pessoa, que é 1,76, deve usar **MatrizPesoAltura[1][9] = 1.760;**

String - não existe de fato um tipo especial para texto, ou seja, qualquer texto é interpretado como uma sequência, ou um vetor de elementos tipo caractere. No entanto, a linguagem C permite que vetores de caracteres possam ser expressados de forma mais compacta. Assim, a declaração **const char vetc[6]= {'T', 'e', 'x', 't', 'o', '\0'};** pode ser feita como **const char vetc[] = "Texto";**, considerando que o compilador põe automaticamente o caractere especial '\0' ao fim do vetor, o que indica o fim da **string**, além de aumentar o tamanho da *string* para 6.

Ponteiros - esse, geralmente, é um tópico que traz muitas confusões iniciais para os estudantes, não é essencial para um programa embarcado, mas traz muitos benefícios para a programação com manipulação de matrizes. Um ponteiro é uma variável, mas que não armazena um valor, mas sim um endereço de memória, onde está o valor que deve ser acessado. Apesar da linguagem C tratar os endereços das variáveis de forma transparente para o programador, vamos considerar que a variável do tipo **unsigned int Idade** esteja no endereço de memória de dados 0x0100, e que o valor atual dessa variável seja 28. Você pode acessar esse valor durante o programa através da transferência direta **VarAprocessar = Idade;**. No entanto, você também pode acessar esse valor de maneira indireta, através de um ponteiro. Primeiro deve-se declarar um ponteiro para o tipo unsigned int, compatível com a variável para a qual ele "apontará", e deve ser usado o prefixo "*" para indicar ao compilador que esta é uma variável do tipo ponteiro: **unsigned int *PontIdade**. Considere que este ponteiro está no endereço de memória de dados 0x0102. Vale ressaltar que o símbolo asterisco está associado ao nome da variável, e não ao tipo, portanto as duas variáveis poderiam ter sido declaradas juntas: **unsigned int Idade, *PontIdade;**. Quando é criado, o ponteiro não possui um valor específico, e deve ser carregado posteriormente pelo programador com o endereço da variável a ser apontada, descrito

pelo nome da variável precedida de "&". Como dito, o programador não manipulará diretamente o valor do endereço, mas, para o nosso exemplo, nós sabemos que o valor 0x0100, e não 28, será transferido para o PontIdade em: **PontIdade = &Idade;**. Interpreta-se: a variável ponteiro PontIdade recebe o endereço da variável Idade, ou melhor, a partir de agora a variável PontIdade **aponta** para a variável Idade. Isso significa que o valor de Idade pode ser acessado indiretamente pelo ponteiro, através do símbolo "*". É o mesmo usado para declarar uma variável ponteiro, mas, quando não está em uma declaração, assume um papel diferente, o que gera confusão para muitos programadores. Este símbolo é usado antes da variável ponteiro, indicando que o valor a ser transferido não é o que está dentro da variável ponteiro, mas sim o que está na variável apontada pelo ponteiro. Dessa forma, para transferir o valor de Idade (28) indiretamente através do ponteiro, já atualizado, usamos o comando **VarAprocessar = *PontIdade;**. Interpreta-se: a variável VarAprocessar recebe o valor da variável apontada pelo ponteiro PontIdade. Note que se o símbolo asterisco não fosse usado, o valor transferido para VarAprocessar seria 0x0100, que é o valor que está contido na PontIdade. também que, se por engano fosse usado o símbolo "&" ao invés de "*", o valor transferido para VarAprocessar seria 0x0102, que é o valor do endereço da PontIdade. Isso pode ser visto na Figura 2.11:

Figura 2.11 | Valores na memória e consequência de comandos em C para o exemplo



Fonte: elaborada pelo autor.

Os ponteiros são muito utilizados na manipulação de matrizes, e por isso, quando se deseja fazer um ponteiro apontar para um vetor, geralmente para o primeiro elemento, ao invés de **PontChar = &Vetchar[0];**, podemos usar **PontChar = Vetchar;**, que é a mesma

coisa. Devemos notar também que, de fato, matrizes e ponteiros são da mesma natureza, ou seja, podem ter os nomes usados um no lugar do outro, se o ponteiro estiver apontando para o primeiro elemento da matriz, é claro. A grande diferença é que o ponteiro é solto, no sentido de poder apontar para elementos diferentes durante o programa, e a matriz não. Seria um "ponteiro constante". Dessa forma, depois de PontChar apontar para Vetchar[0], é possível usar Pontchar[2] ao invés de Vetchar[2].

Exemplificando

É verdade que para este caso anterior o uso do ponteiro não parece muito útil, foi apenas para informar as suas propriedades. Vejamos agora um bom exemplo de utilização do ponteiro. Considere uma rotina para transmitir uma mensagem de texto, que pode ser escolhida entre três opções, através de um canal serial. O registrador fictício **BufferTX** é usado para enviar os dados serialmente via hardware, e o registrador **FLAGBufferLivre** é usado para indicar que a última transmissão já terminou e o canal está livre para a transmissão de um novo caractere. O programa apresentado na Figura 2.12 pode ser usado para isso.

Figura 2.12 | Trecho de programa para transmitir caracteres

```
#include <avr/io.h>       //inclui nomes dos regs
#define TRUE  1
#define TamMsg  24

int main(void){       //indica início do código
     const char Msgs[3][TamMsg] = {"Inicializando sistema";
                                   "Microcontroladores AVR";
                                   "Kroton rede de Educacao"};
     const char *PontMsg;
     unsigned int CntMsg = 0, i;

     while(TRUE){     //loop infinito
          PontMsg = Msgs[CntMsg]; //ponteiro aponta para Msg atual
                                  // equivalente a PontMsg = &Msgs[CntMsg][0];
          for(i=0; i<TamMsg; i++, PontMsg++){  //loop TamMsg vezes
               if(*PontMsg=='\0') break;              //encerra se encontrar fim da string
               while(!FLAGBufferLivre){ }       //aguarda fim  da transmissão anterior
               BufferTX = *PontMsg;             //carrega caractere a ser transmitido
          }
          if(CntMsg==2) CntMsg = 0;        //atualiza CntMsg
          else CntMsg++;
     }
}
```

Fonte: elaborada pelo autor.

Devemos nos atentar para a manipulação aritmética dos ponteiros. Quando se incrementa um ponteiro, o seu valor não é simplesmente aumentado em uma unidade, e sim na quantidade de bytes que correspondem ao tipo de variável que ele aponta. Por exemplo: *float *pont; pont++;*. O ponteiro será aumentado em 4, ou seja, vai apontar para a próxima variável desse tipo. Além das matrizes, os ponteiros também são úteis para acessar estruturas, com o operador "->".

Pesquise mais

O uso das estruturas e ponteiros em linguagem C permitem manipulação de formas avançadas de dados, o que pode servir para a criação de estruturas de dados como "árvores". Procure entender melhor como isso é feito. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/binst.html> e <https://www.ime.usp.br/~pf/algoritmos/aulas/bint.html>. Acesso em: 20 jun. 2017.

O typecasting de ponteiros deve ser escrito sempre que necessário. Segue a forma:

char Vet[3]; unsigned char *Pont = (char *)Vet;

Estruturas

As estruturas de dados são semelhantes às matrizes, pois agrupam vários elementos. No entanto, esses elementos não precisam ser do mesmo tipo, podendo, inclusive, criar estruturas dentro de estruturas. As estruturas são de fato um novo "tipo" de variável, criado pelo programador, e pode ser um conjunto de outras variáveis de qualquer tipo. Desse modo, o programador deve analisar, pela natureza do problema, quais devem ser os tipos das variáveis que comporão o conjunto, ou melhor, a estrutura. Por exemplo, se é um programa que manipulará alguns valores de características físicas e pessoais, como nome, peso, altura, ano de nascimento, idade e CPF, pode ser definida a **struct CadastroPessoal**, visto na Figura 2.13:

Figura 2.13 | Exemplo do uso de estrutura de dados

```
#include <avr/io.h>

struct CadastroPessoal{
   char nome[80];
   float peso;
   float altura;
   unsigned short int idade;
   unsigned int AnoNasc;
   unsigned long int CPF;
}

void main(void){

   struct CadastroPessoal Joao, Andre;

   Joao.nome = "Joao Gomes Silva";
   Joao.peso = 72.8;
   Joao.altura = 1.77;
   Joao.idade = 26;
   Andre.idade = Joao.idade;
   ...
```

Fonte: elaborada pelo autor.

Typedef - serve para redefinir novos tipos de dados. Pode ser usado para encurtar/abreviar nomes compostos de variáveis. Por exemplo, depois do comando: **typedef const unsigned char cteUchar;**, você pode declarar variáveis do tipo citado como: **cteUchar Letra = 'a';**. Também pode ser usado para não precisar escrever "*struct*" toda vez que for declarar uma variável a partir de estrutura. Assim, a variável declarada na Figura 2.13 **struct cadastroPessoal Joao;** poderia ser reescrita como tcadastroPessoal Joao;, em conjunto com a redefinição **typedef struct cadastroPessoal tCadastroPessoal;**, que deve vir antes. Atenção! Um ponteiro que foi declarado a partir de uma estrutura existente, ao ser incrementado em 1, será adicionado ao valor correspondente ao tamanho da estrutura, em bytes. Apesar disto ser transparente para o programador, que apenas executa o Pont++ para o ponteiro apontar para a próxima estrutura na memória, em alguns casos é importante saber o tamanho de uma estrutura, que pode ser obtido pelo comando sizeof(nomedaEstrutura);. Ou mesmo: **unsigned int a = sizeof(float); //a←4.**

Funções

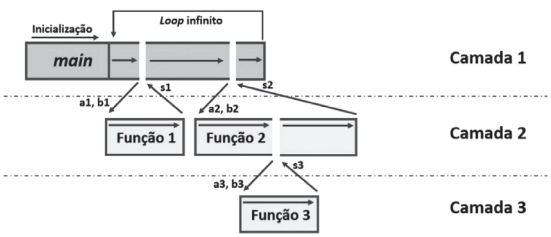
As funções são usadas para modularizar o programa, ou seja, dividi-lo em partes para que o problema completo seja resolvido a

partir de tarefas menores, o que é muito recomendado para a criação de sistemas mais complexos. Além disso, o uso de funções permite reaproveitamento de código, para que uma mesma tarefa que é usada em várias partes do programa não precise ser reescrita. Uma função em programação possui os mesmos princípios das funções matemáticas. Descrever o comportamento de uma grandeza, que é a saída da função, em função de uma ou mais variáveis de entrada. Quando se define uma função se define qual é o tipo da variável de saída e qual é o nome da função, além de quais são as variáveis de entrada, seus nomes e tipos. Apesar de em alguns casos haver a necessidade de calcular mais de um valor de saída, cada função pode retornar um único valor. No entanto, existem duas formas de contornar essa questão: usando uma *struct* como variável de saída, assim todos os campos da estrutura podem ser retornados pela função. Outra forma é alterar o valor de variáveis acessadas globalmente, o que ficará mais claro logo adiante.

Assimile

Os programas criados através de funções são chamados de modularizados, e podem ser vistos como um processo em camadas, onde cada camada corresponde a um grupo de funções. Dessa forma, como todos os programas se iniciam na função ***main***, dizemos que ela é a primeira camada, e todas os comandos diretamente escritos na *main* estão nessa mesma camada. Quando a função *main* "chama" ou "invoca" alguma outra função dizemos que o programa "desceu" para uma camada abaixo, e permanece nesta até que retorne para a camada superior, ou seja, termine seu trabalho e volte para a *main*, ou invoca uma terceira função, desviando o programa para uma camada mais abaixo. Como pode ser visto no fluxograma da Figura 2.14:

Figura 2.14 | Fluxograma para um programa modularizado



Fonte: elaborada pelo autor.

Podemos perceber que uma função pode também não possuir valores de retorno ou de entrada, ou que é caracterizado pelo indicador ***void***, que significa "vazio" ou "ausente". Por isso, é comum que nos programas embarcados utilizem a função principal declarada como: **void main(void){**, pois não é invocada por nenhuma outra (é a "raiz", ou a primeira camada), e não precisa de nenhum valor de entrada e também de saída. No entanto, alguns compiladores utilizam o padrão genérico da linguagem C: **int main(void){**, mesmo que não retorne nada, como o que usaremos.

A sintaxe padrão para a declaração de uma função é a seguinte, para duas entradas:

TipoVariavelSaida Nomedafunção(TipoVarEntr1 NomeVarEntr1, TipoVarEntr1 NomeVarEntr2){

CorpoDaFunção;

Return ValordeSaida;

}

Exemplificando

Assim, para criar um programa modularizado simples, podemos usar o exemplo para calcular a média de um conjunto numérico, com 2 elementos.

Figura 2.15 | Exemplo de programa modularizado equivalente à Figura 2.14

```
#include <avr/io.h>       //inclui nomes dos regs
#define TRUE  1
int funcao1(int a1, int b1); //declara funções
int funcao2(int a2, int b2); //que serão usadas
int funcao3(int a3, int b3);

void main(void){       //indica início do código
   int a, b, media1, media2; //declara vars
   a = 10;       //atribui valor arbitrário p teste
   b = 20;
   while(TRUE){     //loop infinito
      media1 = funcao1(a, b); //1ª maneira
      media2 = funcao2(a, b); //2ª maneira
   }
}
```

```
int funcao1(a1, b1){ //descreve funções
   int s1; //declara variável local
   s1 = (a1 + b1)/2;
   return s1; //retorna saída
}

int funcao2(a2, b2){
   int s2;
   s2 = funcao3(a2, b2);
   s2 = s2/2;
   return s2;
}

int funcao3(a3, b3){
   int s3;
   s3 = a3 + b3;
   return s3;
}
```

Fonte: elaborada pelo autor.

Quando uma função invoca outra que possui parâmetros de entrada, ele "passa" os valores das entradas que devem ser processados pela segunda função, mas não os transfere, e sim, os copia. Considere, por exemplo, que a função *main* invoca uma função e passa a variável Var1 com o valor 10. Mesmo se a função invocada alterar o valor dessa variável, ao retornar à função principal, a variável conterá o valor antigo, sem a alteração, pois essa alteração foi feita em uma cópia, e não na variável original da *main*.

Escopo

Essa palavra pode parecer estranha, mas é muito importante na análise de programas. Cada elemento do programa, como variáveis, funções e até definições possui um escopo associado a ele, ou seja, qual é a região do programa em que ele está acessível, ou a que ele pertence. Dessa forma, se uma variável é criada dentro de uma função, ela só poderá ser acessada dentro daquela função, pois é considerada uma **variável local** daquela função. Nem mesmo as funções que estiverem camadas abaixo poderão acessar variáveis que foram declaradas em funções acima que a invocaram. Para criar **variáveis globais**, a declaração deve ser feita acima e fora da função *main*, e estas podem ser manipuladas por qualquer parte do programa. Apesar de não ser como o comando **GOTO**, que não deve ser usado em nenhuma ocasião, é recomendado que o uso de variáveis globais seja evitado, exceto em algumas situações bem específicas. Isso se deve ao fato de que, se uma variável ou região da memória de dados seja manipulada por várias funções do programa, aumentam as chances de conter valores indevidos, que foram alterados por outras funções, quando uma certa função for tratar esses dados. Por isso, é recomendado que cada função manipule exclusivamente os dados nela tratados, considerando as cópias que foram transferidas como parâmetro de entrada. As variáveis globais são geralmente usadas para trocar informações entre a função *main* e as rotinas de tratamento de interrupções, o que será estudado na próxima unidade.

Reflita

O que deve acontecer se existirem variáveis globais e locais com o mesmo nome? Sabendo que a linguagem C permite isso, o que deve

acontecer em um programa assim quando for executado? As variáveis se sobrepõem, ou são tratadas em separado?

Modularização em arquivos

Quando se trata de um programa com poucas funções, e que são pequenas, a escrita pode ser feita em um único arquivo. Caso contrário, é recomendado que o programa, além de modularizado em funções, seja também escrito a partir de múltiplos arquivos. Isso melhora o desenvolvimento do programa, pois cada parte é tratada individualmente, permitindo abstrair vários outros detalhes que são do projeto, mas não são relevantes naquele trecho. Também melhora a legibilidade, pois um arquivo de programa que contenha muitas linhas causa mais dificuldade ler o programa e encontrar partes específicas para análise. Podemos ver na Figura 2.16 como o exemplo da Figura 2.15 pode ser reescrito através de três arquivos:

Figura 2.16 | Exemplo anterior modularizado em três arquivos

main.c

```
#include <avr/io.h>
#include <Funcoes.h>
#define TRUE  1

void main(void){
   int a, b, media1, media2;
   a = 10;
   b = 20;
   while(TRUE){
      media1 = funcao1(a, b);
      media2 = funcao2(a, b);
   }
}
```

Funcoes.h

```
int funcao1(int a1, int b1);
int funcao2(int a2, int b2);
```

Funcoes.c

```
#include <Funcoes.h>

int funcao1(a1, b1){
   int s1;
   s1 = (a1 + b1)/2;
   return s1;
}

int funcao3(a3, b3){
   int s3;
   s3 = a3 + b3;
   return s3;
}

int funcao2(a2, b2){
   int s2;
   s2 = funcao3(a2, b2);
   s2 = s2/2;
   return s2;
}
```

Fonte: elaborada pelo autor.

Sem medo de errar

Vamos agora resolver a questão desta seção. Lembre-se de que nessa situação-problema, uma empresa de automação residencial contatou você como o responsável técnico da área de

projetos de eletrônica e automação. Inicialmente, você recebeu a responsabilidade de elaborar um algoritmo básico para o controle de um portão de garagem, através de um fluxograma. Em seguida, deve fazer um programa em C, que atue e controle o portão de forma eficiente, a partir das três entradas: botão de acionamento (abre e fecha) - BOT, sensores fim de curso alto - SA, e baixo - SB (detecta portão fechado). Saídas: um relé para subir - RS, e outro para descer - RD. Além disso, você conta com mais dois sensores digitais, um detecta a presença do veículo abaixo do portão - SP, para evitar colisão, e o outro detecta sobrecarga (ou sobrecorrente) do motor - SS, indicando bloqueio mecânico. Também foi adicionada uma saída ligada a uma lanterna sinalizadora - LS, que deve piscar em 0,5 Hz quando o portão estiver em movimento ou aberto, e deve ser desligada 5 segundos após o fechamento do portão. O portão agora deve fechar de forma automática, caso o dono tenha esquecido, 10 segundos depois de permanecido aberto, sem risco de danificar o carro, é claro. Se durante o acionamento houver sobrecarga, o motor deve parar, a lanterna piscará por 5 segundos em 2 Hz, antes de desligar, e o portão deve voltar para sua posição anterior, até completar o curso e parar. Para ter referência de tempo, você deve usar a função genérica "_delay_ms(X)". Não se esqueça que, durante o movimento, o sensor final não pode ficar longos períodos (>50 milissegundos) sem ser verificado, pois se este for acionado e isso não for detectado o quanto antes pelo sistema, o motor ficará ligado e a parte mecânica será danificada. Inicialmente, recordaremos como foi a solução da situação-problema anterior, pois vamos aproveitar o que foi feito lá. Foi utilizada uma máquina de estados, com três abordagens diferentes. A ideia era simplesmente ligar o motor para subir o portão, caso o botão fosse pressionado, e desligar quando o sensor fim de curso alto for atingido. E para fechar o portão, a mesma coisa. Liga para baixo quando o botão for pressionado, e desliga o motor quando o sensor baixo for detectado. Agora, nessa nova situação, vamos tratar algumas considerações importantes, que podem ocorrer durante o funcionamento do conjunto que deve estar preparado. Por exemplo, na inicialização do sistema, é sensato pensar que o portão pode ter sido desligado sem terminar seu curso, entreaberto. Dessa forma, o programa, quando iniciado, pode checar se o portão não está completamente fechado com: **if(SB)**, e, se não estiver, deve acionar o portão até fechá-lo. Devemos também considerar os sensores de

presença SP e de sobrecarga SS, para não causar nenhum dano físico. Se o SP acusar presença, o sistema deve desligar o motor e aguardar que o objeto saia de baixo do portão, religando o motor novamente. Isso pode ser feito através do comando de controle **while(SP){ }**, ou seja, não faz nada enquanto SP==1. Se o SS acusar sobrecarga em alguma situação do programa, este deve ser desviado para a função que trata da lanterna. Assim, a função deve saber qual é o sentido do movimento do portão, para cima ou para baixo, para ligar novamente o portão e verificar se a sobrecarga permanece, mesmo depois dos 5 segundos em que a lanterna é acionada. O restante do programa, ou seja, o loop infinito pode ser implementado como o programa apresentado para a situação-problema anterior. A diferença é que agora o SS deverá ser sempre verificado quando o motor estiver acionado, e tratado com a função citada, e o SP deve ser checado quando o portão estiver descendo, ou seja fechando, conforme dito anteriormente.

Figura 2.17 | Possível solução para a situação-problema

```
#include <avr/io.h>
#include <util/delay.h>

#define TRUE  1
#define EstAUSENTE  0
#define EstVERIFPRESENCA 1
#define EstPRESENCA 2

#define DESLIGADO 0
#define LIGADO 1


void main(void){
   unsigned int i, VarEST = EstAUSENTE;
   unsigned int i VarSP, cnt, cntaux;
   DDRA = 0x00;   //config PortA entrada: SP
   DDRB = 0xFF;   // config PortB saída: LUZ
   PORTB = DESLIGADO;    // desliga a LUZ

while(TRUE){
     VarSP = PINA;
     switch(VarEST){
     case EstAUSENTE:
       if(VarSP){  //se detectou pode ter gente
          VarEST = EstVERIFPRESENCA;
          cnt = 300;  //atualiza para contagem de 3seg
          cntaux = 50; //carrega para contar 0.5 seg
       }
       break;
     case EstVERIFPRESENCA:
       if(!cntaux){//se contou 0.5 seg -> n tem gente
          VarEST = EstAUSENTE;
          PORTB = DESLIGADO;    //desliga LUZ
       }else cntaux--;
       if(VarSP) cntaux = 50; //se SP=1 recarrega cntaux
       if(!cnt){  //se contou 3 seg aqui -> tem gente
          VarEST = EstPRESENCA ;
          PORTB = LIGADO ;    //liga LUZ
          cnt = 1000;    //carrega para contar 10seg
       }else cnt--;
       break;
     default:
       if(!cnt){  //aguarda 10 s
          VarEST = EstVERIFPRE
          cnt = 300; //carrega p
          cntaux = 50; //carrega
       }else cnt--;
       break;
     }
    _delay_ms(10);
   }
}
```

Fonte: elaborada pelo autor.

Avançando na prática

Acionamento inteligente de uma lâmpada

Descrição da situação-problema

Sua empresa de automação foi contratada para elaborar um sistema de automação de uma lâmpada na área externa de

um condomínio, a partir de um sensor de presença. Você foi incumbido a desenvolver o software desse sistema, que possui apenas uma entrada, um sensor de presença - **SP**, que estará conectado à porta A, e uma saída, o relé para ligar a lâmpada - **RL**, que é acionado pela porta B do ATmega328. Você descobriu que na verdade os sensores disponíveis não são de presença, mas de movimento, ou seja, se uma pessoa permanecer imóvel o sensor não detectará sua presença. O síndico do condomínio lhe passou as seguintes especificações: a ideia é desligar a lâmpada quando não houver ninguém na área externa com sinuca, para economizar energia, mas a lâmpada não deve ser ligada quando pessoas passarem na calçada em frente ao bar, o que pode ser detectado no SP. Por isso, quando o SP detectar movimento, o sistema deve verificar se o movimento será novamente detectado nos próximos instantes, o que confirma alguém jogando sinuca, e não um pedestre passando na calçada. Desse modo, o sistema deve permanecer por 3 segundos em uma condição de testar, filtrando o sinal de presença. Nessa condição, se for detectado um intervalo de mais de 0,5 segundo contínuo sem ocorrer movimento, o processo deve descartar a presença de alguém, mas se durante os 3 segundos isso não for detectado, alguém está na área e a lâmpada deve ser acesa. Depois disso, a lâmpada deve permanecer acesa incondicionalmente durante 10 segundos, e em seguida o sistema decide se deve apagar ou deixar a lâmpada acesa, a partir da mesma condição citada. Com base nessas especificações descritas, uma outra equipe está responsável por construir o hardware e as instalações, e você deve construir um programa em C que controle corretamente o acionamento da lâmpada.

Resolução da situação-problema

Para resolver essa questão, é interessante recorrer ao clássico e universal método da máquina de estados, com três estados: AUSENTE, VERIFICAPRESENCA e PRESENCA. No primeiro, a lâmpada deve estar desligada, pois o sistema considera que a área externa com sinuca está vazia. Nesse estado, se o sistema detectar movimento, pode ser alguém passando na rua ou alguém que chegou na área externa. A partir daí o programa vai para o

segundo estado, onde vai verificar se a condição será satisfeita para ligar a luz. Para sincronizar as tarefas, podemos usar uma função **delay_ms(10);** ao fim do loop infinito, e usar contadores para ter referência temporal. Dessa forma, considerando que o loop infinito será executado a aproximadamente cada 10 ms, pois o tempo de execução do programa pode ser desprezado, podemos usar dois contadores, um com contagem de 300, chamado **cnt**, para contar os 3 segundos e outro, **cntaux**, com 50 para contar o 0,5 segundo. Não devemos usar apenas um contador, porque o primeiro contará continuamente os 3 segundos corridos e o outro será reatualizado diversas vezes, pois será responsável por indicar o intervalo de tempo entre duas ocorrências de movimento consecutivas. Ainda nesse estado, os contadores devem permanecer contando até algum dos dois contadores "estourar", ou seja, atingir seu limite de contagem. Se ocorrer primeiro com o cntaux, significa que passou 0,5 segundo sem o sensor detectar movimento, e, portanto, é um falso alarme. O programa deve então voltar para o primeiro estado, sem acender a lâmpada. Se o *cnt* estourar antes significa que os 3 segundos se passaram e o **cntaux** não estourou nesse período. Isso apenas acontece se o *cntaux* for reatualizado, retornando sua contagem a cada vez que o sensor SP detectar movimento. Assim, se alguém estiver se movimentando no ambiente, o sensor SP gerará vários pulsos de sinal seguidos, com curtos intervalos. Isso fará com que o programa reinicie a contagem do *cntaux* muitas vezes durante os três segundos, e o *cntaux* não estourará. Quando os três segundos se passarem, com a continuidade do movimento detectada, o conjunto deve acionar a lâmpada e aguardar por 10 segundos. Em seguida, o teste dos 3 segundos é realizado novamente, mas sem desligar a saída, pois a(s) pessoa(s) ainda pode(m) estar lá. Assim, se nesse intervalo for detectado um período maior que 0,5 segundo sem o SP atuar (*cntaux* estourar), a lâmpada deve ser desligada e o sistema volta para o primeiro estado. Caso contrário, o sistema mantém a saída acionada, seguindo para o terceiro estado, e assim por diante.

Figura 2.18 | Possível programa como solução do problema proposto

```
#include <avr/io.h>
#include <util/delay.h>

#define TRUE  1
#define EstAUSENTE  0
#define EstVERIFPRESENCA 1
#define EstPRESENCA 2

#define DESLIGADO 0
#define LIGADO 1


void main(void){
   unsigned int i, VarEST = EstAUSENTE;
   unsigned int i VarSP, cnt, cntaux;
   DDRA = 0x00;   //config PortA entrada: SP
   DDRB = 0xFF;   // config PortB saída: LUZ
   PORTB = DESLIGADO;    // desliga a LUZ

while(TRUE){
     VarSP = PINA;
     switch(VarEST){
     case EstAUSENTE:
       if(VarSP){  //se detectou pode ter gente
          VarEST = EstVERIFPRESENCA;
          cnt = 300;  //atualiza para contagem de 3seg
          cntaux = 50; //carrega para contar 0.5 seg
       }
       break;
     case EstVERIFPRESENCA:
       if(!cntaux){//se contou 0.5 seg -> n tem gente
          VarEST = EstAUSENTE;
          PORTB = DESLIGADO;   //desliga LUZ
       }else cntaux--;
       if(VarSP) cntaux = 50; //se SP=1 recarrega cntaux
       if(!cnt){  //se contou 3 seg aqui -> tem gente
          VarEST = EstPRESENCA ;
          PORTB = LIGADO ;   //liga LUZ
          cnt = 1000;    //carrega para contar 10seg
       }else cnt--;
       break;

     default:
       if(!cnt){  //aguarda 10 s
          VarEST = EstVERIFPRE
          cnt = 300; //carrega p
          cntaux = 50; //carrega
       }else cnt--;
       break;
     }
   _delay_ms(10);
   }
}
```

Fonte: elaborada pelo autor.

Faça valer a pena

1. A linguagem de programação C, assim como outras, é composta por um conjunto de regras de sintaxe e semântica, assim como as linguagens humanas. Dessa forma, quando um programador desenvolve um código em linguagem C, ele deve conhecer e respeitar essas regras para que o programa possa ser compilado e funcione conforme o esperado.

Sobre a programação de microcontroladores em linguagem C, observe as seguintes afirmações:

I – Quando a função Main() invoca uma outra função, **void Func1(int a, intb);**, por exemplo, não há a necessidade de usar valores para os parâmetros de entrada "**a**" e "**b**", pois existe o indicador "void", que significa "ausência".
II – A função principal de qualquer programa embarcado pode possuir qualquer nome, e é considerada, para a abstração de programa em camadas, a função raiz, ou seja, a primeira camada. Todas as outras funções que forem invocadas pela principal também estarão na mesma camada que a raiz, mas não podem retornar valores, apenas recebê-los para processamento.
III – Todas as funções em C podem receber inúmeros parâmetros de entrada, porém, todos esses dados devem ser do mesmo tipo. A única exceção é quando se usam matrizes como sinais de entrada de uma função.
IV – As funções como na matemática possuem apenas um retorno, que para a programação foi estabelecido a possibilidade de retornar dois valores além de um, unicamente. Não existe nenhuma outra maneira da função

invocada retornar mais valores de saída para a função que a chamou, para este caso deve ser criada outra função distinta, mesmo que o seu "corpo" (interior) seja o mesmo.
Considerando a veracidade das afirmações, qual das alternativas representa a sequência correta?

a) V, V, F, F.
b) F, F, F, F.
c) V, F, V, F.
d) F, F, V, F.
e) F, F, F, V.

2. Uma das facilidades da programação em linguagem C é possibilitar o manuseio de endereços das variáveis, através dos ponteiros.

Considere a construção de um programa embarcado em linguagem C para o ATmega328, onde a variável tipo ponteiro, unsigned int *PontVet1;, deve ser usada para acessar o terceiro elemento do vetor unsigned int Vet1[10];. Utilizando a variável unsigned int Var1;, qual das alternativas a seguir representa a sintaxe e os comandos corretos para esta ação?

a) **Var1 = Vet1[*PontVet1+3];**, o que significa que o conteúdo do elemento do vetor Vet1 indicado pelo índice PontVet1 será transferido para a **Var1**.
b) **PontVet1 = *Vet1[2]; Var1 = &PontVet1;**. O primeiro comando serve para fazer o ponteiro apontar para o terceiro elemento do veto. O seguinte busca o conteúdo do endereço apontado por **PontVet1** e carrega para a variável **Var1**.
c) **PontVet1 = Vet1; PontVet1+=2; Var1 = *PontVet1;**. O primeiro comando serve para fazer o ponteiro apontar para o primeiro elemento do vetor. O segundo comando incrementa o ponteiro em dois endereços, fazendo ele apontar para o terceiro elemento do vetor. O último transfere o conteúdo do endereço apontado por **PontVet1** para a variável **Var1**.
d) **PontVet1 = &Vet1[3]; PontVet1++; Var1 = PontVet1;**. O primeiro comando serve para fazer o ponteiro apontar para o terceiro elemento do vetor. O segundo comando incrementa o ponteiro para ele apontar para o próximo elemento do vetor. O seguinte busca o conteúdo do endereço apontado por **PontVet1** e carrega para a variável **Var1**.
e) **PontVet1 = Vet1[3]; Var1 = &PontVet1;**. O primeiro comando serve para fazer o ponteiro apontar para o terceiro elemento do vetor. O último comando transfere o conteúdo do endereço apontado por PontVet1 para a variável **Var1**.

3. Na linguagem de programação C, além das definições que podem ser feitas com a diretiva **#define**, existem outras formas de se criar amarrações de valores ou tipos de variáveis.

Que utilidade o comando "*typedef*" apresenta na programação em linguagem C?

a) O uso do *typedef* é obrigatório para todos os programas embarcados, assim como o loop infinito, uma vez que é responsável por definir todos os tipos básicos das variáveis que serão manipuladas pelo programa.
b) O *typedef* é usado exclusivamente para estruturas de dados, pois é responsável por associar os ponteiros e as estruturas que serão apontadas.
c) Esse comando especial tem como finalidade criar tipos de dados universais, que podem ser utilizados na criação de outros programas, através de atualizações distribuídas pela internet.
d) O *typedef* serve para a manipulação de matrizes, quando estas possuem mais de duas dimensões. A partir daí é necessário que esse tipo composto de dados seja representado através de estruturas, definidas pelo *typedef*.
e) Este comando serve para redefinir novos tipos de dados. Pode ser usado para encurtar/abreviar nomes compostos de variáveis. Por exemplo, depois do comando: **typedef const unsigned char cteUchar;**, você pode declarar variáveis do tipo citado como: **cteUchar Letra = 'a';.**

Seção 2.3

Ambiente de trabalho e simulação

Diálogo aberto

Você deve estar ansioso para começar a colocar em prática os conhecimentos adquiridos. Calma! Esse momento chegou, enfim. Nesta seção, estudaremos todos os passos para colocar um programa escrito em linguagem C para ser executado na placa Arduino UNO. Vamos apresentar o programa que será usado como ambiente de trabalho, o AtmelStudio, criado pela própria fabricante do microcontrolador ATmega328, a Atmel. Depois de configurar o software AVRDude para descarregar o código para o Arduino, através do AtmelStudio, criaremos um simples programa de teste (pisca o Led) para transferi-lo para a placa, provavelmente o momento mais empolgante desse curso, quando a teoria ganha vida, se torna real!

Depois que você implementar e executar esse programa básico, estará pronto para criar programas mais complexos para serem descarregados na placa, como os estudados anteriormente por nós, mesmo ainda sem conhecer os periféricos. Para consolidar esse estudo, vamos resolver as últimas questões da situação-problema desta unidade. Nela, uma empresa de automação residencial consolidada no mercado contrata você como o engenheiro de projetos de eletrônica e automação. Assim que ingressou, você recebeu a responsabilidade de elaborar um algoritmo básico para o controle de um portão de garagem. Você já elaborou o fluxograma e o programa para uma situação mais simplificada na primeira seção. Depois, você criou um programa mais elaborado, considerando todos os detalhes pertinentes para o controle avançado de um portão de garagem, com o uso de funções. Agora é a parte final do projeto, que é a prototipagem. Logicamente, não é necessário construir um sistema completo, em um portão de garagem de verdade, mas podemos descarregar o mesmo programa em uma placa, e usar 5 chaves e dois leds para representar as entradas do sistema e os dois relés de saída, respectivamente. Apenas para o botão de acionamento deve ser usado um botão do tipo *push button* (não retentivo). As

demais entradas, que são os sensores, são representadas por chaves retentivas, que mantêm o último estado imposto (chaves retentivas).

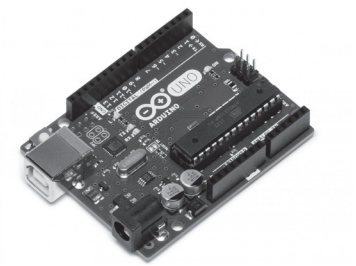
Não pode faltar

Agora que já estudamos todos os aspectos fundamentais dos microcontroladores, e em específico o ATmega328, estamos prontos para criar os primeiros experimentos práticos para um sistema embarcado.

Arduino

Como dito anteriormente, a placa que usaremos é a Arduino UNO, que se assemelha a um tutorial, mostrando passo a passo como o seu computador pessoal será configurado devidamente para podermos usar a placa com sucesso. Você pode usar alguma das placas que estão disponíveis no laboratório da sua faculdade, que deve ser feito através do seu professor. Além disso, se você se interessar pelo assunto, placas como estas são bem acessíveis para ter em casa, para você iniciar seu próprio projeto. A placa Arduino UNO pode ser vista na Figura 2.19:

Figura 2.19 | Placa Arduino UNO



Fonte: <https://goo.gl/B6fCXn>. Acesso em: 31 maio 2017.

Afinal de contas, o que significa Arduino? E qual é a sua relação com a AVR, ou Atmel? De fato, estes não são a mesma coisa, ou seja, não pertencem ao mesmo fabricante. O Arduino é um projeto *open source* (fonte aberta), ou aberto para uso e isento de direitos autorais, que surgiu na Itália com o objetivo de criar uma plataforma de programação de sistemas embarcados, onde muitos detalhes de projeto de hardware e software são "pré-construídos", o que torna o processo de prototipagem muito mais rápido. Acontece

que os inventores da plataforma Arduino escolheram, entre outros concorrentes, os microcontroladores AVR, fabricados pela empresa Atmel, para compor as suas placas, e o modelo ATmega328P para o modelo UNO, a mais popular. Apesar de existir muitos exemplos variados prontos para serem compilados e transferidos para placa - o que é ótimo para atingir resultados mais rápidos, partindo de um ponto já feito e funcionando - o nosso foco aqui não é trabalhar sobre a plataforma Arduino. O objetivo do curso é estudar os aspectos e a programação de microcontroladores e microprocessadores, e utilizaremos a estrutura do Arduino, que se tornou acessível para aplicar esses conceitos e construir sistemas reais e completos, ao invés de simplesmente pegar prontos, sem saber como foram feitos.

Configuração do AtmelStudio

Devemos ressaltar também que o único sistema operacional abordado será o Windows, e se você possui outro, como Linux ou IOS, deve procurar seguir as instruções, que são muito semelhantes a estas, no site oficial. Disponível em: <http://www.atmel.com/tools/atmelavrtoolchainforlinux.aspx>. Acesso em: 17 jul. 2017.

Antes de conectar a placa ao seu computador, instale os dois aplicativos seguintes, que serão utilizados para criar e transferir nossos programas para a placa.

O Arduino IDE: <https://www.arduino.cc/en/Main/Software>. Acesso em: 17 jul. 2017.

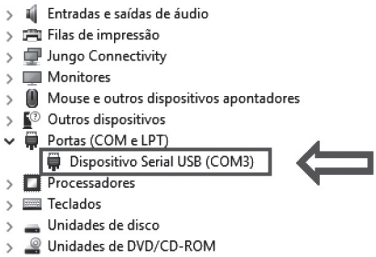
E o AtmelStudio: <http://www.atmel.com/tools/atmelstudio.aspx#download>. Acesso em: 17 jul. 2017.

O primeiro é a IDE - *Integrated Development Environment*, ou ambiente de desenvolvimento integrado, composto pelo editor de programas e compilador da Arduino, que não serão utilizados por nós para criar programas. Aproveitaremos apenas alguns arquivos e programas presentes em sua pasta de instalação para realizar a gravação do microcontrolador através do software Atmel Studio, através da sua ferramenta AVRDude, que será integrada ao AtmelStudio. Esse segundo software é o ambiente de desenvolvimento criado pela própria fabricante do microcontrolador ATmega328, como se pode perceber no nome. Deve-se levar em consideração a versão do aplicativo disponível, que atualmente em 2017 é a sétima, mas pode ser incrementada com o tempo, e apresentar pequenas alterações.

Configuração do AtmelStudio7

Depois que estiver instalado os dois programas citados, conecte a sua placa Arduino UNO ao computador através do cabo USB. O Arduino se comunica com o computador pessoal através de um canal serial, ou seja, o computador "enxerga" o Arduino como uma porta serial, igual àquelas que existiam nos antigos PCs, usadas para conectar teclados e outros. Obviamente, essa será uma porta serial virtual, uma vez que está sendo emulada através de uma porta USB. Todas as portas seriais criadas nos PCs, em particular no Windows, recebem um número identificador. Para descobrir qual número a sua placa UNO recebeu, vá em →**Painel de controle**, em seguida →**hardware e sons** e depois → **gerenciador de dispositivos**. Verifique qual é a porta COM como na Figura 2.20, que nesse caso é a porta COM3:

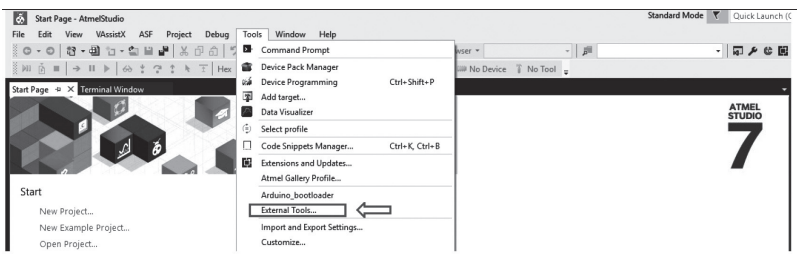
Figura 2.20 | Gerenciador de dispositivos do Windows



Fonte: elaborada pelo autor.

Agora que você tem essa informação, abra o programa AtmelStudio7. Depois de aberto, clique na opção de aba → ***tools*** (ferramentas) e depois em → ***external tools...***, conforme mostra a Figura 2.21:

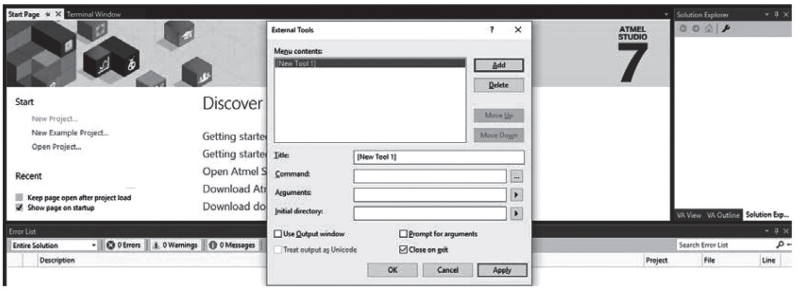
Figura 2.21 | Adicionando ferramenta externa ao AtmelStudio7



Fonte: elaborada pelo autor.

Agora, para adicionar uma ferramenta, clique no botão Add (adicionar), como mostra a Figura 2.22:

Figura 2.22 | Adicionando ferramenta externa ao AtmelStudio7



Fonte: elaborada pelo autor.

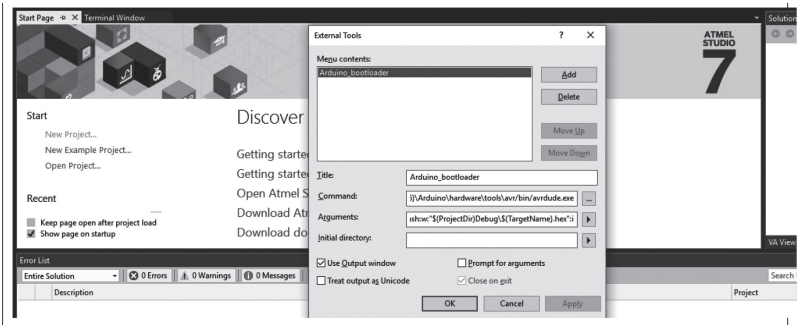
Depois de abrir o Atmelstudio7, você deve preencher os campos conforme os passos a seguir, os quais se referem a comandos técnicos muito específicos, que não serão explicados nesta seção por não apresentarem relevância.

1. No campo ***Title***, ou título, escreva "Gravador Arduino", que será o nome usado para a ferramenta externa. Se quiser, altere o nome como preferir.
2. No campo ***Command***, ou comando, escreva a sequência: **C:\Program Files (x86)\Arduino\hardware\tools\avr/bin/avrdude.exe.** Note que para este caso, o *Windows* está na versão em inglês. Caso o seu esteja em português, altere o começo da sequência de *Program Files* para "Arquivos de Programas". Verifique também, antes de concluir, se o AVRDude está mesmo instalado no local indicado pelo endereço.
3. No campo ***Arguments***, ou argumentos, escreva: **-C"C:\Program Files (x86)\Arduino\hardware\tools\avr/etc/avrdude.conf" -v –p atmega328p -carduino –PCOM3 -b115200 -D –U flash:w: "$(ProjectDir)Debug\$(TargetName).hex":i.** Novamente, repare que está em inglês e precisa ser alterado, caso necessário, assim como no campo Command. Repare também que você deve ajustar o seu número da PCOM, de acordo com o valor verificado em alguns passos atrás.

4. Deixe habilitada a opção "***Use Output Window***", no canto inferior esquerdo.
5. Clique em Ok.

Depois de completar os campos conforme indicado, sua tela deve ficar igual à Figura 2.23:

Figura 2.23 | Preenchimento dos campos para o *AVRdude*



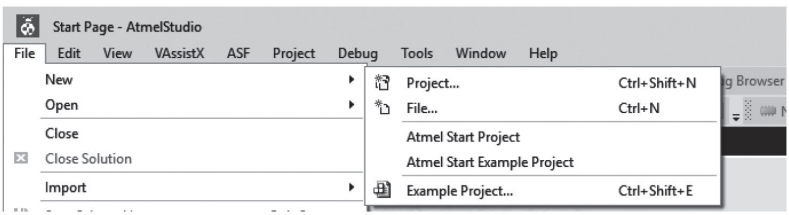
Fonte: elaborada pelo autor.

Agora tudo está configurado e pronto para funcionar, falta apenas criar um projeto, fazer um programa e descarregar na placa para ver funcionando.

Criando um projeto

Para criar um novo projeto, clique em → **File,** → **New,** → **Project**, conforme a Figura 2.24:

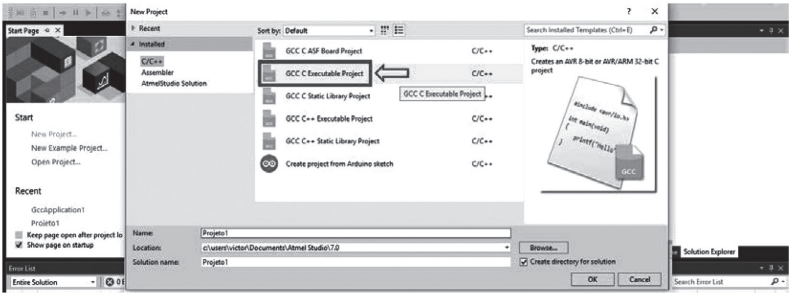
Figura 2.24 | Criando um projeto novo



Fonte: elaborada pelo autor.

Agora, escolha o tipo de projeto, que para nós será sempre o GCC C, conforme a Figura 2.25:

Figura 2.25 | Especificando o tipo de projeto a ser criado



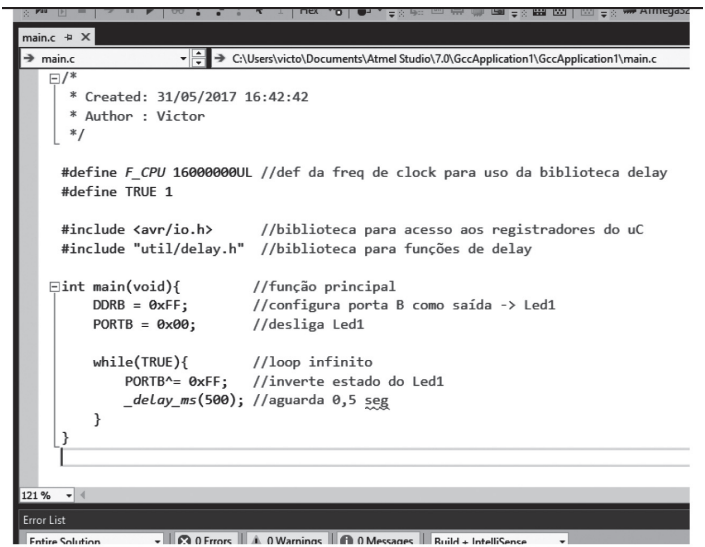
Fonte: elaborada pelo autor.

Em seguida, escolha o modelo do microcontrolador que receberá o programa compilado, que no nosso caso é o ATmega328P. Depois disso, o AtmelStudio lhe entrega um programa esqueleto com apenas a função principal, a inclusão do cabeçalho e o loop infinito.

Exemplificando

Vamos criar agora um exemplo bem básico para visualizar o seu efeito na placa, através do seu Led. Preencha a sua main.c como na Figura 2.26.

Figura 2.26 | Primeiro programa exemplo a ser compilado



Fonte: elaborada pelo autor.

Observe todos os comandos do programa, e perceba que todos eles já foram estudados por nós nas seções anteriores. Apesar de já ter sido mencionado, a função de *delay* (atraso), que simplesmente "congela" o processamento durante alguns instantes, para ser usada na prática, depende do uso de uma biblioteca específica do AtmelStudio, e de uma definição antes.

Assimile

Não esqueça de incluir o arquivo de ***delay***, que já possui as funções de *delay* prontas, precisando apenas que seja definida antes no programa a frequência de relógio que será utilizada, através do indicador F_CPU.

Finalmente, para descarregar o programa na sua placa UNO, ou melhor, fazer o download do programa, clique em ***Tools*** e em seguida em "Gravador Arduino", ou o nome que você tenha escolhido para a ferramenta externa. Quando clicado, o programa compilado é descarregado para a placa, que começa a executá-lo logo em seguida.

Reflita

Depois de rodar o código pronto, faça algumas mudanças nele para ver o comportamento, como alterar o valor de delay para 50 ou 5000 ms.

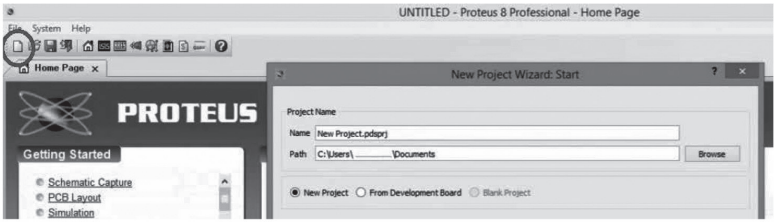
Lembre-se de salvar o programa antes de compilar, ou configure para o AtmelStudio fazer isso de forma automática.

Proteus ISIS

Agora vamos configurar o ambiente de simulação para nossos projetos. Inicialmente, baixe e instale a versão gratuita do programa na primeira opção (Proteus professional demonstation) do site: <https://www.labcenter.com/downloads>.

Depois de abrir o programa, clique em →***File,*** →***New Project Wizard***, (ou no ícone destacado na Figura 2.27). Escolha um nome para o projeto e um local (pasta de arquivos) no seu computador.

Figura 2.27 | Especificando o tipo de projeto a ser criado

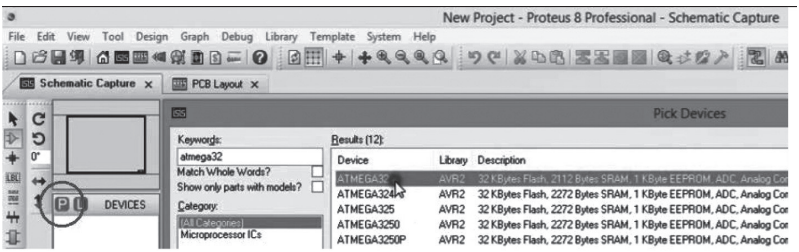


Fonte: elaborada pelo autor.

Clique em *next*. Depois disso, selecione a segunda opção "*Create a schematic...*". Escolha a opção para o tamanho da área do seu esquemático, ***Lanscape A2***, por exemplo. Clique em *next*. Nessa próxima tela não há mudança, uma vez que a criação de placas de circuito impresso não é o nosso foco. Portanto, selecione "***Do note create PCB layout***" e clique em next. Agora também não altere, mantenha a escolha: "*no firmware Project*". Clique em next. Confira e clique em *Finish*.

Agora, clique no ícone "P", destacado pelo círculo na Figura 2.28. No campo de pesquisa, no canto superior esquerdo da janela "*Pick Devices*", digite **ATmega32** e escolha o modelo do dispositivo (pode ser com ou sem o "8P" no final), conforme mostra a Figura 2.28:

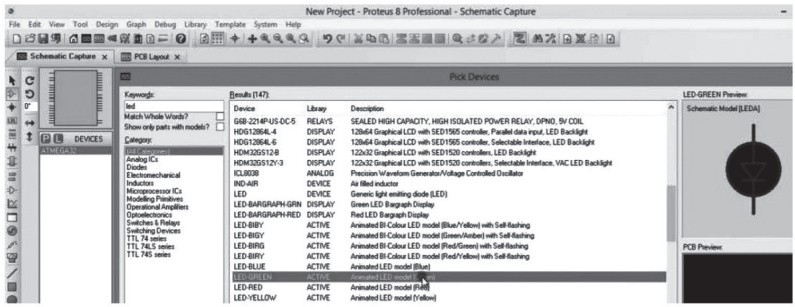
Figura 2.28 | Escolha do modelo do microcontrolador usado no projeto



Fonte: elaborada pelo autor.

Clique em qualquer lugar na área para inserir o chip no projeto. Clique novamente no "P", e agora escreva **led**. Pode ser qualquer um, como o da Figura 2.29:

Figura 2.29 | Adicionando leds ao projeto

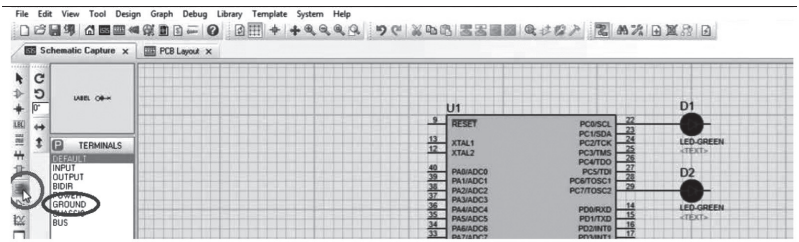


Fonte: elaborada pelo autor.

Antes de inserir no projeto, é possível rotacionar a posição do *led*, logo acima do botão "P", ao lado esquerdo da imagem geral do projeto. Insira dois *leds* ao lado do microcontrolador, e os conecte nas portas clicando nos terminais que devem ser conectados.

Após isso, precisamos prover um sinal de GND (*ground*), ou sinal TERRA. Para isso, clique no ícone no canto esquerdo, "Terminals Mode", destacado com um círculo na Figura 2.30, e em seguida em GROUND, destacado com uma elipse. Insira um sinal GND e o conecte aos *leds* e ao chip:

Figura 2.30 | Especificando o tipo de projeto a ser criado

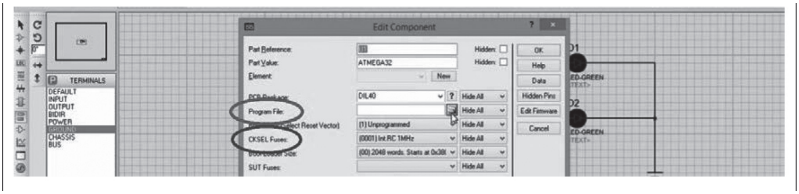


Fonte: elaborada pelo autor.

Botões e resistores podem ser adicionados pelo mesmo processo. Agora, para carregar o Atmega32 com o programa que compilamos, dê duplo clique no chip que aparece no projeto. Você deve clicar no símbolo de pastas no campo ***Program File*** (destacado na Figura 2.31) e encontrar no explorador de arquivos o programa (.hex) que foi gerado no projeto do AtmelStudio, geralmente em C:\Users\nome\Documents\Atmel Studio. Outro campo deve ser ajustado, o **CKSEL**

fuses:, também destacado na Figura 2.31). Se não quiser adicionar cristais, utilize uma fonte de relógio interna, o **IntRC_8MHz**, por exemplo. Clique em Ok.

Figura 2.31 | Embarcando o programa e selecionando a fonte de relógio



Fonte: elaborada pelo autor.

Por fim, depois de tudo devidamente conectado e configurado, clique na seta de "*play*", no canto inferior esquerdo da tela principal (em azul), e aguarde por até uns 5 segundos. Se não houver erros no procedimento ou no código, o modelo deve representar o processo corretamente. Agora você pode simular todos os exercícios e exemplos que vimos, além dos seus próprios projetos, para encontrar e tratar os possíveis erros que aparecem. A partir de agora não vamos tratar apenas de termos teóricos, pois os periféricos são muito úteis em aplicações reais, e já temos as ferramentas para simular e implementar de fato sistemas mais simples.

Pesquise mais

Para saber um pouco mais sobre plataforma de desenvolvimento de projetos com Arduino da Atmel, consulte o site. Disponível em: <https://www.embarcados.com.br/atmel-studio/>. Acesso em: 26 jun. 2017.

Sem medo de errar

Vamos agora resolver a questão desta seção. Na situação-problema desta unidade, uma empresa de automação residencial contrata você como o engenheiro de projetos de eletrônica e automação. Você recebeu a responsabilidade de elaborar um algoritmo básico para o controle de um portão de garagem. Depois de elaborar o fluxograma com o algoritmo necessário, você implementou a solução em um programa embarcado. Primeiro foi adotada uma forma simplificada para o problema, para servir de base

para o projeto final, incluindo detalhes para um controle avançado de portão de garagem. Chegou a hora de simular e também emular esse sistema. Simular virtualmente em um programa de simulação, e a palavra emular significa construir na prática, mas que funciona de maneira equivalente, não igual. Isso se dá pelo fato de que não automatizaremos um portão verdadeiro, mas descarregaremos o código no Arduino e "emular" o processo através de entradas e saídas figurativas, chaves e *leds*. Isso significa que se todo o resto da parte externa for construído de fato em um portão de garagem verdadeiro, o sistema funcionará corretamente. O seu funcionamento deve respeitar as seguintes exigências técnicas de projeto de um portão de garagem, que atue e controle o portão de forma eficiente, a partir das três entradas: botão de acionamento (abre e fecha) - BOT, sensores fim de curso alto - SA, e baixo - SB (detecta portão fechado). Saídas: um relé para subir - RS, e outro para descer - RD. Além disso, você conta com mais dois sensores digitais, um detecta a presença do veículo abaixo do portão - SP, para evitar colisão, e o outro detecta sobrecarga (ou sobrecorrente) do motor - SS, indicando bloqueio mecânico. Também foi adicionada uma saída ligada a uma lanterna sinalizadora - LS, que deve piscar em 0,5 Hz quando o portão estiver em movimento ou aberto, e deve ser desligada 5 segundos após o fechamento do portão. O portão agora deve fechar de forma automática, caso o dono tenha esquecido, 10 segundos depois de permanecido aberto, sem risco de danificar o carro, é claro. Se durante o acionamento houver sobrecarga, o motor deve parar, a lanterna piscará por 5 segundos em 2 Hz, antes de desligar, e o portão deve voltar para sua posição anterior, até completar o curso e parar. Para ter referência de tempo, você deve usar a função genérica "_delay_ms(X)". Não se esqueça que, durante o movimento, o sensor final não pode ficar longos períodos (>50 milissegundos) sem ser verificado, pois se este for acionado e isso não for detectado o quanto antes pelo sistema, o motor ficará ligado e a parte mecânica será danificada. Inicialmente, vamos recordar como foi a solução da situação-problema anterior, pois vamos aproveitar o que foi feito lá. Foi utilizada uma máquina de estados, com três abordagens diferentes. A ideia era simplesmente ligar o motor para subir o portão, caso o botão fosse pressionado, e desligar quando o sensor fim de curso alto for atingido. E para fechar o portão, a mesma coisa. Liga para baixo quando o botão for

pressionado, e desliga o motor quando o sensor baixo for detectado. Agora, nesta nova situação, vamos tratar algumas considerações importantes, que podem ocorrer durante o funcionamento do conjunto, que deve estar preparado. Por exemplo, na inicialização do sistema, é sensato pensar que o portão pode ter sido desligado sem terminar seu curso, entreaberto. Dessa forma, o programa, quando iniciado, pode checar se o portão não está completamente fechado com: **if(SB)**, e, se não estiver, deve acionar o portão até fechá-lo. Devemos também considerar os sensores de presença SP e de sobrecarga SS, para não causar nenhum dano físico. Se o SP acusar presença, o sistema deve desligar o motor e aguardar que o objeto saia de baixo do portão, religando o motor novamente. Isso pode ser feito através do comando de controle **while(SP){ }**, ou seja, não faz nada enquanto SP==1. Se o SS acusar sobrecarga em alguma situação do programa, este deve ser desviado para a função que trata da lanterna. Assim, a função deve saber qual é o sentido do movimento do portão, para cima ou para baixo, para ligar novamente o portão e verificar se a sobrecarga permanece, mesmo depois dos 5 segundos em que a lanterna é acionada. O restante do programa, ou seja, o loop infinito pode ser implementado como o programa apresentado para a situação-problema anterior. A diferença é que agora o SS deverá ser sempre verificado quando o motor estiver acionado, e tratado com a função citada, e o SP deve ser checado quando o portão estiver descendo, ou seja fechando, conforme dito anteriormente.

Figura 2.32 | Possível solução para a situação-problema

```
#include <avr/io.h>
#include <util/delay.h>

#define TRUE  1
#define EstAUSENTE  0
#define EstVERIFPRESENCA 1
#define EstPRESENCA 2

#define DESLIGADO 0
#define LIGADO 1


void main(void){
   unsigned int VarEST = EstAUSENTE;
   unsigned int VarSP, cnt, cntaux;
   DDRA = 0x00;   //config PortA entrada: SP
   DDRB = 0xFF;   // config PortB saída: LUZ
   PORTB = DESLIGADO;    // desliga a LUZ

while(TRUE){
     VarSP = input(PINA);
     switch(VarEST){
     case EstAUSENTE:
       if(VarSP){  //se detectou pode ter gente
          VarEST = EstVERIFPRESENCA;
          cnt = 300;   //atualiza para contagem de 3seg
          cntaux = 50; //carrega para contar 0.5 seg
       }
       break;
     case EstVERIFPRESENCA:
       if(!cntaux){//se contou 0.5 seg -> n tem gente
          VarEST = EstAUSENTE;
          PORTB = DESLIGADO;    //desliga LUZ
       }else cntaux--;
       if(VarSP) cntaux = 50; //se SP=1 recarrega cntaux
       if(!cnt){  //se contou 3 seg aqui -> tem gente
          VarEST = EstPRESENCA ;
          PORTB = LIGADO ;    //liga LUZ
          cnt = 1000;     //carrega para contar 10seg
       }else cnt--;
       break;

     default:
       if(!cnt){  //aguarda 10 segs
          VarEST = EstVERIFPRESENCA ;
          cnt = 300; //carrega p 3 seg
          cntaux = 50;  //carrega p 0.5
       }else cnt--;
       break;
     }
    _delay_ms(10);
    }
}
```

Fonte: elaborada pelo autor.

Avançando na prática

Automação de uma lâmpada inteligente

Descrição da situação-problema

Vamos agora emular o problema de "avançando na prática" da Seção 2.2. Nela, sua empresa de automação foi contratada para elaborar um sistema de automação de uma lâmpada na área externa de um bar, a partir de um sensor de presença. Você foi incumbido a desenvolver o software desse sistema, que possui apenas uma entrada, um sensor de presença - **SP**, que estará conectado à porta A, e uma saída, o relé para ligar a lâmpada - **RL**, que é acionado pela porta B do ATmega328. Você descobriu que na verdade os sensores disponíveis não são de presença, mas de movimento, ou seja, se uma pessoa permanecer imóvel, o sensor não detectará sua presença. O dono do bar passou as seguintes especificações: a ideia é desligar a lâmpada quando não houver ninguém na área externa com sinuca, para economizar energia, mas a lâmpada não deve ser ligada quando pessoas passarem na calçada em frente ao bar, o que pode ser detectado no SP. Por isso, quando o SP detectar movimento, o sistema deve verificar se o movimento será novamente detectado nos próximos instantes, o que confirma alguém jogando sinuca, e não um pedestre passando na calçada. Desse modo, o sistema deve permanecer por 3 segundos em uma condição de testar, filtrando o sinal de presença. Nessa condição, se for detectado um intervalo de mais de 0,5 segundo contínuo sem ocorrer movimento, o processo deve descartar a presença de alguém, mas se durante os 3 segundos isso não for detectado, alguém está na área e a lâmpada deve ser acesa. Depois disso, a lâmpada deve permanecer acesa incondicionalmente durante 10 segundos, e em seguida o sistema decide se deve apagar ou deixar a lâmpada acesa, a partir da mesma condição citada. Faça um programa em C que controle corretamente o acionamento da lâmpada conforme as especificações.

Resolução da situação-problema

Para resolver essa questão, é interessante recorrer ao clássico e universal método da máquina de estados, com três estados:

AUSENTE, VERIFICAPRESENCA e PRESENCA. No primeiro, a lâmpada deve estar desligada, pois o sistema considera que a área externa com sinuca está vazia. Nesse estado, se o sistema detectar movimento, pode ser alguém passando na rua ou alguém que chegou na área externa. A partir daí o programa vai para o segundo estado, onde vai verificar se a condição será satisfeita para ligar a luz. Para sincronizar as tarefas, podemos usar uma função **delay_ms(10)**; ao fim do loop infinito, e usar contadores para ter referência temporal. Dessa forma, considerando que o loop infinito será executado a aproximadamente cada 10 ms, pois o tempo de execução do programa pode ser desprezado, podemos usar dois contadores, um com contagem de 300, chamado cnt, para contar os três segundos e outro, **cntaux**, com 50 para contar o 0,5 segundo. Não devemos usar apenas um contador, porque o primeiro contará continuamente os 3 segundos corridos e o outro será reatualizado diversas vezes, pois será responsável por indicar o intervalo de tempo entre duas ocorrências de movimento consecutivas. Ainda nesse estado, os contadores devem permanecer contando até algum dos dois contadores "estourar", ou seja, atingir seu limite de contagem. Se ocorrer primeiro com o *cntaux*, significa que passou 0,5 segundo sem o sensor detectar movimento, e, portanto, é um falso alarme. O programa deve então voltar para o primeiro estado, sem acender a lâmpada. Se o *cnt* estourar antes significa que os 3 segundos se passaram e o *cntaux* não estourou nesse período. Isso apenas acontece se o *cntaux* for reatualizado, retornando sua contagem a cada vez que o sensor SP detectar movimento. Assim, se alguém estiver se movimentando no ambiente, o sensor SP gerará vários pulsos de sinal seguidos, com curtos intervalos. Isso fará com que o programa reinicie a contagem do *cntaux* muitas vezes durante os três segundos, e o *cntaux* não estourará. Quando os três segundos se passarem, com a continuidade do movimento detectada, o conjunto deve acionar a lâmpada e aguardar por 10 segundos. Em seguida, o teste dos 3 segundos é realizado novamente, mas sem desligar a saída, pois a(s) pessoa(s) ainda pode(m) estar lá. Assim, se nesse intervalo for detectado um período maior que 0,5 segundo sem o SP atuar (*cntaux* estourar), a lâmpada deve ser desligada e o sistema volta para o primeiro estado. Caso contrário, o sistema mantém a saída acionada, seguindo para o terceiro estado, e assim por diante.

Figura 2.33 | Possível programa como solução do problema proposto

```
#include <avr/io.h>
#include <util/delay.h>

#define TRUE  1
#define EstAUSENTE  0
#define EstVERIFPRESENCA 1
#define EstPRESENCA 2

#define DESLIGADO 0
#define LIGADO 1


void main(void){
   unsigned int VarEST = EstAUSENTE;
   unsigned int VarSP, cnt, cntaux;
   DDRA = 0x00;   //config PortA entrada: SP
   DDRB = 0xFF;   // config PortB saída: LUZ
   PORTB = DESLIGADO;    // desliga a LUZ
while(TRUE){
     VarSP = Input(PINA);
     switch(VarEST){
     case EstAUSENTE:
       if(VarSP){  //se detectou pode ter gente
          VarEST = EstVERIFPRESENCA;
          cnt = 300;  //atualiza para contagem de 3seg
          cntaux = 50; //carrega para contar 0.5 seg
       }
       break;
     case EstVERIFPRESENCA:
       if(!cntaux){//se contou 0.5 seg -> n tem gente
          VarEST = EstAUSENTE;
          PORTB = DESLIGADO;    //desliga LUZ
       }else cntaux--;
       if(VarSP) cntaux = 50; //se SP=1 recarrega cntaux
       if(!cnt){  //se contou 3 seg aqui -> tem gente
          VarEST = EstPRESENCA ;
          PORTB = LIGADO ;    //liga LUZ
          cnt = 1000;     //carrega para contar 10seg
       }else cnt--;
       break;
     default:
       if(!cnt){  //aguarda 10 segs
          VarEST = EstVERIFPRESENCA ;
          cnt = 300; //carrega p 3 seg
          cntaux = 50;  //carrega p 0.5
       }else cnt--;
       break;
     }
   _delay_ms(10);
   }
}
```

Fonte: elaborada pelo autor.

Faça valer a pena

1. Todas as funções escritas na linguagem C devem sempre mostrar os tipos e os nomes dos parâmetros de entrada e de saída, no momento da declaração, descrição e invocação. Mesmo quando não existe nenhum parâmetro de entrada ou de saída, este deve ser indicado pelo termo *void*. Analisando os microcontroladores AVR programados em linguagem C, por que a função main do AtmelStudio precisa do retorno ***int*** ao invés de ***void***, apesar de não retornar nada, e para ninguém?

a) Porque esse programa em específico retorna um valor inteiro para o compilador, no momento da compilação, indicando que o programa foi escrito corretamente e está pronto para ser descarregado no microcontrolador.

b) Porque a Atmel, para o AtmelStudio, escolheu utilizar o padrão universal para a linguagem C, onde a *main* retorna um valor para o sistema operacional. Como feito nos computadores, a empresa resolveu manter esse padrão, mesmo sem utilizar o comando de retorno ***return XX***; ao final da função *main*, o que não seria nunca executado por causa do loop infinito.

c) Porque todos os programas embarcados são obrigados a manipular pelo menos uma variável do tipo inteiro, primordial para os processos.

d) Na verdade, esse retorno não é necessário, funciona apenas para aumentar a legibilidade do programa, mas pode ser omitido sem nenhum problema.

e) O termo ***int*** mostra que o tipo de variável mais complexo que será usado pelo programa será o inteiro. Assim, o programa pode manipular dados ***char*** sem problemas. No entanto, se o programa manipular dados do tipo flutuante, o termo ***int*** deve ser substituído por ***float.***

2. As memórias usadas internamente aos microprocessadores e microcontroladores são diferentes e apresentam utilidades específicas. Existem as memórias de dados, que podem ser voláteis ou não, e a memória de programa.
O programa descarregado no microcontrolador Atmega328 vai para qual de suas memórias internas?

a) Memória *Cache*.
b) Memória EEPROM.
c) Memória FLASH.
d) Registradores de propósito geral.
e) Memória RAM.

3. A linguagem de programação C é muito utilizada para desenvolvimento de programas embarcados, e conta com uma coleção de regras de sintaxe e de semântica, que devem ser conhecidas e respeitadas pelo programador. Para um programa embarcado em linguagem C, desenvolvido para o microcontrolador ATmega328, onde um Led está ligado a uma saída na porta digital B, por que o comando **PORTB^=0xFF**; é responsável por inverter o estado do Led?

a) Porque esse é um comando especial para os microcontroladores da família AVR, e, apesar de não ter nenhuma ligação com operações lógicas, é um comando responsável por inverter os estados da saída, também conhecido em inglês como *toggle*.
b) Porque esse comando representa uma operação lógica AND, e cada vez que ela é executada, o estado da saída do Led é invertido.
c) Na verdade, esse comando não é responsável por inverter o estado das saídas conectadas à porta B, mas, sim, por apenas acionar os seus bits. Dessa forma, esse comando está sempre acionando as saídas, ao invés de ligar uma vez, e desligar na próxima, assim sucessivamente.
d) Esse comando é uma abreviação do comando de atualização do registrador de E/S para a porta B, onde o próximo valor será o resultado do valor atual com a operação XOR (ou exclusivo) com os bits 0b11111111. Essa operação lógica faz com que todos os bits da Porta B sejam invertidos, independentemente dos seus estados.
e) Porque todas as vezes que uma porta configurada como saída digital recebe algum comando com o operador "^", os seus estados são automaticamente alternados para o outro valor binário, o que se mostra muito útil em alguns casos especiais.

Referências

BARNETT, R. H.; COX, S.; O'CULL, L. **Embedded C programming and the Atmel AVR**. 2. ed. Nova York: Delmar Cengage, 2007.

KERNIGHAN, B. W.; RITCHIE, D. M. **The C programming language**. 2. ed. New Jersey: Prentice Hall, 1991.

LIMA C. B. D; VILLAÇA, M. V. M. **AVR e Arduino técnicas de projeto**. Florianópolis: [s.n.], 2012.

MARGUSH, T. S. **Some assembly required**: assembly language programming with the AVR microcontrollers. Nova York: CRC Press, 2011.

MAZIDI M. A.; NAIMI, S.; NAIMI, S. **AVR microcontroller and embedded systems**: using assembly and C. Nova York: Pearson, 2010.

MIZRAHI, V. V. **Treinamento em linguagem C**. Nova Jersey: Pearson, 2008.

STALLINGS, W. **Arquitetura e organização de computadores**. 8. ed. São Paulo: Prentice Hall, 2009.

Periféricos básicos

Convite ao estudo

Entramos agora em uma nova fase do nosso estudo, pois as próximas abordagens não serão apenas teóricas, e todos os exemplos e os problemas tratados poderão ser convertidos em sistemas reais, e por você! Iniciaremos nossa jornada por dentro dos periféricos internos do Atmega328, pelo mais clássico e universal de todos: as portas digitais. Praticamente, todos os projetos embarcados utilizam algum canal digital, seja para um *led*, um botão, uma chave, ou qualquer um dos vários sensores digitais e saídas controladas digitalmente. Depois de entender a sua estrutura, configuração e aplicação, estudaremos um outro periférico muito importante, principalmente para projetos mais avançados, complexos e críticos, que são as interrupções. Veremos como múltiplas rotinas de interrupções podem ser configuradas e gerenciadas pelo programa usuário. Finalizaremos, também, um outro tema muito importante, e que completará nossa abordagem sobre análise e sincronismo temporal de tarefas através dos temporizadores internos, ou timers. Nós já estudamos como sincronizar, através de rotinas de temporização, tarefas paralelas, ou seja, saídas/tratativas que devem ser controladas simultaneamente, com o revezamento adequado do programa. Vimos, inclusive, alguns padrões de algoritmos com contadores para sincronismo primário, que podem ser combinados para prover estruturas mais avançadas. No entanto, agora você perceberá que a utilização dos timers é muito mais elaborada, precisa e prática para o desenvolvimento de sistemas de tempo real. Para aplicar todos esses conhecimentos adquiridos nesta unidade, abordaremos uma situação na qual será necessário construir soluções de hardware e de software, embarcados diretamente através dos elementos estudados.

Nessa situação, você deve desenvolver um dispositivo de interface com o usuário da casa, que será feito em equipe, e sua responsabilidade é o teclado matricial, que identificará para o sistema os comandos numéricos para o controle dos periféricos externos distribuídos pela casa. Como todo projeto profissional, cada responsável faz a sua parte e depois estas são reunidas e sincronizadas por um outro. Você não deve se preocupar com a manipulação dos dados digitados, apenas com a eficiência e a robustez da aquisição, filtro e fornecimento dos dados. Por isso, o seu projeto individual é montar um hardware com um teclado matricial telefônico, um botão, dois leds e um display de 7 segmentos, todos devidamente conectados em um ATmega328 (em um *protoboard* de preferência).

Vamos começar?

Seção 3.1

Portas digitais

Diálogo aberto

Vamos finalmente iniciar nossa jornada nos periféricos, que nos permitem ampliar os horizontes sobre soluções embarcadas. Depois de toda a carga teórica estudada até agora, estamos prontos para criar os primeiros programas e projetos realmente úteis, ou melhor, aplicáveis em problemas de automação recorrentes. Depois de estudar como as portas digitais podem ser configuradas e utilizadas em problemas, você estará apto para desenvolver sistemas de controle digital. O que pode e deve ser explorado por você são todos os periféricos externos que podem ser agregados ao sistema, e como as portas digitais devem ser utilizadas em conjunto. Para começar essa tarefa, abordaremos, ao fim desta seção, uma situação em que você deverá elaborar uma solução para um problema prático. Neste problema, você trabalha em uma equipe para desenvolver um dispositivo de interface com o usuário de uma casa automatizada. Cada um da equipe é responsável por uma parte do projeto de automação doméstica, e a sua tarefa foi desenvolver a interface, tanto em hardware quanto em software, para o teclado matricial, que identificará para o sistema os comandos numéricos. Portanto, você fará agora um sistema "teste", que é autossuficiente, para que depois possa ser incluído no protótipo inicial completo. Sabendo que utilizaremos um teclado numérico matricial para esse projeto, algumas decisões devem ser tomadas para realizar a conexão entre os sinais das teclas e o microcontrolador Atmega328. Inicialmente, você deve decidir o circuito. Para isso, é preciso responder a pergunta: como deve ser a conexão e as configurações dos sinais entre o teclado matricial e o microcontrolador, na relação entre linhas e colunas, para os sinais dos botões serem multiplexados, ao invés de usar um sinal exclusivo para cada botão do teclado? Em seguida, sua tarefa é criar um programa para ser embarcado nesse circuito que você montou, que apresente o valor do botão pressionado no teclado no display de 7 segmentos. Você decidiu, com o chefe

do seu grupo, que o valor escolhido deve permanecer por 1 segundo no display e depois apagar, deixando o sistema pronto para exibir o próximo valor da tecla que venha a ser pressionada pelo usuário. Para as teclas "#" (tralha ou "jogo da velha") e "*" apresente os caracteres "t" e "A", respectivamente (adaptados para o display 7-seg.). As entradas digitais podem apresentar "surtos" em seus sinais nos momentos de transição, alterando seu nível lógico várias vezes antes de se estabilizar no seu valor correto. Este efeito recebe o nome de "*bounce*" (quique, ricochete). As técnicas para combater os efeitos indesejados desse fenômeno são chamadas de "técnicas de debounce", e são filtros para que os comandos de entrada tenham o efeito desejado. Essas técnicas podem ser implementadas em software, hardware, ou em ambos? Para o seu projeto, considerando que a exibição de um valor não é interrompida pelo acionamento de outro botão, e que não se contabiliza quantas vezes cada botão foi apertado, é possível que não haja técnica de "*debounce*" e o funcionamento seja correto? Boa sorte!

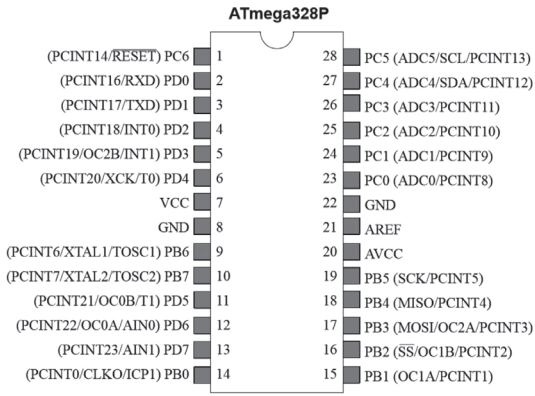
Não pode faltar

Vamos começar nossos estudos pelas portas digitais de entrada e saída, que compõem o periférico interno mais básico, universal e utilizado nas aplicações embarcadas. Ele é responsável por ler entradas digitais, como estado de sensores digitais, chaves, botões ou teclas, bem como acionar saídas digitais, como *leds*, lâmpadas, relés etc.

Todas as portas no μC (abreviação de microcontrolador) ATmega328 possuem funcionalidade de entrada e saída, e podem ser reconfiguradas em tempo de execução, de maneira independente.

As portas digitais são identificadas com as primeiras letras do alfabeto, e nomeadas como PortA, PortB, PortC etc., de acordo com a quantidade de pinos presentes no modelo do μC. No Atmega328, por exemplo, existe da porta B até a porta D, como é possível observar na Figura 3.1:

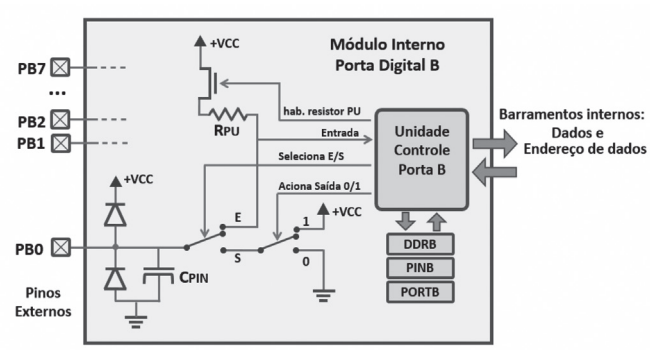
Figura 3.1 | Identificação e mapeamento das portas digitais nos pinos do ATmega328



Fonte: <https://goo.gl/6qJdqz>. Acesso em: 26 set. 2017.

Outra característica importante é a largura de cada porta, que é de 8-bit. Dessa forma, os canais (pinos) de uma mesma porta são identificados por números, de 0 a 7. Por exemplo, o terceiro canal da porta B é identificado como PB2. Cada canal de uma porta pode ser configurado separadamente para atuar a partir de 3 estados diferentes, zero (0V) e um (VCC) quando for uma saída, e alta impedância (Z) quando for configurado como entrada. Pode ser visto na Figura 3.2:

Figura 3.2 | Descrição básica do periférico interno PortB



Fonte: elaborada pelo autor.

Perceba que cada pino digital possui dois diodos de proteção contra surtos, e um capacitor para filtrar transições múltiplas bruscas.

Reflita

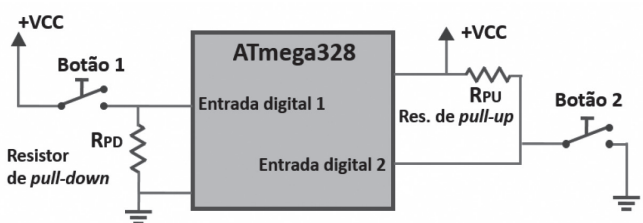
Quando um canal é configurado como entrada (alta impedância), e este não está conectado a nada, qual é o seu potencial elétrico? Ou melhor, a ponta metálica da sua lapiseira, sem tocar em nenhum condutor ou potencial elétrico, apresentará que tensão, se medirmos em relação ao terra universal (solo terrestre)? A primeira ideia que vem à mente é zero, não é? Acontece que isso não é verdade.

Qualquer condutor que esteja desconectado a qualquer potencial elétrico tem o seu próprio potencial indefinido, pois este varia aleatoriamente em função dos campos eletromagnéticos (ondas de rádio) no ambiente, e este sinal não possui potência (energia), uma vez que não há circuito fechado, nem corrente. Dizemos assim que o sinal está "flutuando". Isso significa que, se não impusermos algum potencial nas entradas, o sinal destas flutuará, causará falsas interpretações de acionamentos inexistentes. Por isso, usa-se um resistor, para que o sinal verdadeiro da entrada não seja "mascarado" pelo potencial imposto artificialmente, sem o risco de curto-circuito.

Exemplificando

Podemos perceber claramente aqui como funciona o resistor de pull-down. Se o Botão 1 não está pressionado, o potencial imposto na entrada 1 é 0V (GND), mas se é pressionado, esse valor é alterado para VCC (tensão de alimentação do μC), gerando corrente (inútil) no resistor. Dizemos que esta é uma chave do tipo NA - Normalmente Aberta, pois o estado "normal" (maior parte do tempo) do botão corresponde ao nível lógico zero. De maneira análoga, podemos ver que o mesmo ocorre para o resistor de *pull-up*, de maneira inversa. Assim, podemos dizer que é uma chave do tipo NF - Normalmente Fechada.

Figura 3.3 | Entradas digitais com resistores de *pull-down* e *pull-up*



Fonte: elaborada pelo autor.

Podemos observar na Figura 3.3 que os μCs AVR possuem internamente resistores de *pull-up* individuais para cada entrada, que podem ser habilitados ou não, como veremos logo adiante.

Configurando o periférico Portx

Como vimos na Figura 3.2, existem três registradores presentes em cada porta, que são usados para a configuração do módulo interno feito pelo programa usuário. Sabendo que, assim como as portas, esses registradores são de 8-bit, um canal da porta é configurado de acordo com a posição dos bits acessados nos registradores. Dessa forma, se for necessário configurar o primeiro canal (zero) da porta b, o primeiro bit (posição zero) de cada registrador da **PortB** deve ser configurado, de maneira independente dos demais bits.

DDRx - *Data Direction Register*, ou registrador de direção de dados (da porta x). Responsável por configurar as direções de cada canal da porta. Se o bit de uma posição x do registrador DDRB for 1, significa que o canal correspondente na Porta B é uma saída, e zero, uma entrada (valor padrão na inicialização). Dessa forma, para configurar os seis primeiros canais da Porta B como saídas e os demais como entradas, usa-se no início: **DDRB = 0x3F;**, ou **DDRB = 0b00111111.**

PINx - *Port Input Pins*, ou pinos de entrada da porta x. Nesse registrador é que se encontra os valores dos estados binários de cada pino daquela porta. Considerando que anteriormente a direção dos pinos porta B foi configurada como mencionado, se houver um botão conectado à entrada B7, o seu estado pode ser verificado através da operação: **VarBot = PINB & 0x80;**. Essa é conhecida como uma operação de mascaramento (pois oculta/mascara os dados presentes nos bits 0 até 6 do registrado PINB), que será muito utilizada por nós.

PORTx - *Port Data Register*, ou registrador de dados da porta. Este é o registrador usado para controlar as saídas. Por exemplo, para acionar um *led* que está conectado ao canal PB0 do Atmega328, devidamente configurado como saída (DDRB = 0x01), devemos usar o seguinte comando: PORTB = 0x01.

Assimile

Devido ao fato dos registradores de cada porta serem compartilhados por todos os canais daquela porta, é necessário realizar operações de mascaramento de bits, para que as informações sejam manipuladas individualmente, sem interferências cruzadas. Veremos que esse método pode ser feito com as operações lógicas &, |, ^ e ~, e é usado tanto em configurações, quanto leitura e escrita.

Operações de mascaramento de bits

Este tipo de operação é fundamental sempre que se deseja manipular apenas um (ou alguns) bit de um registrador/variável, sem alterar ou considerar os demais bits. No exemplo anterior, em que foi usado o comando para acionar o *led* no canal PB0, outras saídas na porta B podem ser desativadas.

Acionamento de bit: é feito através de uma operação lógica bit a bit ou entre o registrador que contém o bit a ser acionado, e uma constante chamada "máscara". Esta contém bits 1 apenas nas posições em que se deseja ativar os bits do registrador. Dessa forma, para o exemplo de acionamento do *led*, o adequado é utilizar a operação **PORTB = PORTB | 0x01;**, ou **PORTB |= 0x01;**, para que apenas o primeiro bit (zero) seja acionado. Note que os demais não se alteram.

Zeramento de bit: é feito por uma operação lógica bit a bit E, entre o registrador que contém o bit a ser apagado e uma máscara. A diferença é que, agora, essa máscara contém um bit zero em cada campo correspondente aos bits que se deseja zerar, e bits 1 no resto. Para se desativar uma saída, desligar o *led*, por exemplo, deve-se usar o comando **PORTB &= 0xFE;**, ou mesmo **PORTB &= (~0b00000001);** (inverte todos os bits antes). Esta operação também pode ser usada para se testar apenas um (ou alguns) bits de um registrador. Por exemplo, se uma chave NA estiver conectada ao pino PB3, esta pode ser verificada através de **if(PINB & 0b00001000){ }.**

Analogamente, se esta mesma chave for normalmente fechada, se usa **if(!(PINB & 0x08)){ }.**

Inversão de bit: se não se conhece o estado da saída, e o objetivo é apenas invertê-la, usamos **PORTB ^= 0x01;**. No entanto, alternativamente, os bits dessa porta também podem ser invertidos por uma outra forma. Intuitivamente, o registrador PINx não deve ser

escrito, pois é usado para representar os estados das entradas. No entanto, para os μC AVR, se o valor 1 é escrito em alguma posição desse registrador, o bit correspondente (mesma posição) no registrador PORTx é invertido. Dessa forma, o *led* pode ser invertido, também conhecido como operação de *toggle* (alternância): **PINB ^= 0x01;**. Estes registradores são mapeados e acessados no espaço de memória de dados, e não precisam ser verificados por nós programadores, uma vez que usamos os seus nomes definidos como seus endereços no arquivo incluso "avr/io.h".

Configuração do resistor de *Pull-Up* interno

Não existem registradores adicionais para configurar os resistores internos, pois são aproveitados os registradores descritos acima. Quando um canal é configurado como entrada, o B1, por exemplo (DDRB = 0b11111101;), o segundo bit do registrador PORTB não terá utilidade, pois serve para acionar saídas. Dessa forma, esse campo é aproveitado para configurar os resistores de *pull-up* (não aceito em saídas). Por exemplo, para habilitar o resistor de *pull-up*, na entrada B1, onde há um botão conectado, usamos: **DDRB = 0b11111101; PORTB = 0x02;.**

Podemos observar o resumo sobre as configurações dos possíveis estados do canal genérico n da porta x na Tabela 3.1:

Tabela 3.1 | Possíveis estados de um canal digital genérico do ATmega328

DDRxn PORTxn	0	1
0	Entrada e alta impedância	Saída 0
1	Entrada e resistor de *pull-up* interno	Saída 1

Fonte: elaborada pelo autor.

Transição de canal de entrada para saída

Antes do programa usuário alternar a configuração de um canal digital de entrada ({DDRxn, PORTxn} = 0b00) para saída em nível alto ({DDRxn, PORTxn} = 0b11), um estado intermediário deve ocorrer, que pode ser qualquer um dos restantes, ou seja, os estados: entrada com *pull-up* interno ({DDRxn, PORTxn} = 0b10) ou saída acionada

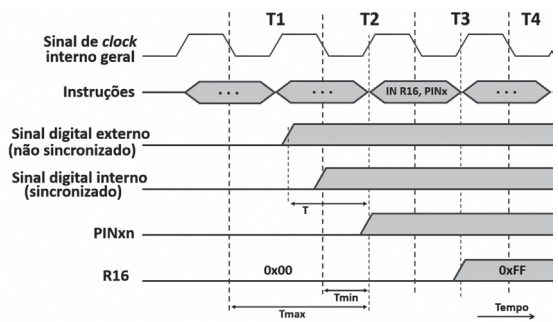
em zero ({DDRxn, PORTxn} = 0b00) deve ser usado como transição para ir do estado entrada (alta impedância) para saída em nível um, diretamente seguidos. Se o programador negligenciar essa condição, não haverá problemas de compilação e nem defeito do circuito. O que pode ocorrer é o acionamento não ser efetivado, e a saída não ser ligada como deveria. Outra situação semelhante que é tratada da mesma maneira, é quando se deseja trocar entre os estados "entrada com *pull-up* interno" para "saída com acionamento zero". Algum dos dois outros possíveis estados devem ser utilizados como passagem, ou estado intermediário.

Leitura do valor de um pino

A leitura do estado do canal n de uma porta digital x deve ser feita através do registrador PINx, bit n, independentemente da configuração do canal como entrada ou saída, feita no registrador de E/S DDRx, bit n. No entanto, os valores digitais externos (nível no pino externo do microcontrolador) não são imediatamente atualizados no registrador de leitura, pois esse sinal passa por um *latch* ("trava eletrônica"), para entrar em sincronismo com o relógio do sistema. Isso é muito importante para se evitar metaestabilidade, que é quando a transição do sinal ocorre muito próximo à transição do sinal de clock, que pode causar falsas interpretações. Isso também provoca um pequeno atraso no sinal, na maioria, desprezível para as nossas aplicações. No entanto, é interessante saber como esse processo ocorre para situações mais críticas.

A Figura 3.4 mostra um diagrama sobre a sincronização temporal de um sinal externo digital, usado como entrada.

Figura 3.4 | Diagrama temporal da amostragem de um canal digital externo



Fonte: elaborada pelo autor.

Apesar de se tratar de um único canal (pino), o registrador recebe o valor 0xFF, apenas para ser genérico. Se fosse escolhido um canal específico, o bit 1 da porta B por exemplo, o último sinal do gráfico remeteria ao segundo bit do registrador PINB. Podemos notar também, na Figura 3.4, que T é o intervalo de tempo entre o acionamento real da entrada (sinal externo) e a sua devida detecção, que é feita internamente pelo programa através do registrador PINx. Os valores máximos e mínimos para o **T** também são destacados.

Leitura de um sinal digital interno

Em algumas ocasiões, um sinal de saída pode ser utilizado como entrada (realimentação), ou seja, algum teste de decisão depende do seu estado. No entanto, existe uma restrição entre acionamentos e leituras digitais consecutivas: é necessário que exista uma instrução NOP (*no operation*, ou sem operação) entre estas duas ações. Por exemplo, se o registrador R16 possui 0xFF, e a porta B (saída) e o R17 possuem 0x00, não é possível escrever na porta com o R16, e ler seu valor logo em seguida para o R17, pois esta ação não ocorrerá efetivamente (leria 0x00 ao invés do valor carregado 0xFF).

Exemplificando

Vamos descrever agora uma situação arbitrária para a configuração da porta B: os canais 0 e 1 serão saída em nível alto; os canais 2 e 3 como saídas em nível baixo, 4 e 5 entradas em alta impedância e 6 e 7 entradas com resistores internos de *pull-up*.

Figura 3.5 | Exemplo de configuração da Porta B do ATmega328

Linguagem *Assembly*

```
...
LDI   R16, (1<<PB7)|(1<<PB6)|(1<<PB1)|(1<<PB0)
LDI   R17, 0x0F
OUT   PORTB, R16     ; PORTB ← 0b11000011;
OUT   DDRB,  R17     ; DDRB  ← 0b00001111;
NOP                  ; aguarda 1 ciclo p sincronizar
IN    R16, PINB      ; faz a leitura do valor correto
...
```

Linguagem C

```
...
unsigned char i ;

PORTB = 0b11000011;
DDRB = (1<<DDB3) | (1<<DDB2)
       | (1<<DDB1) | (1<<DDB0); //0b00001111;
_delay_us(1);  // aguarda  para sincronizar
i = PINB;          // faz a leitura do valor correto
...
```

Fonte: elaborada pelo autor.

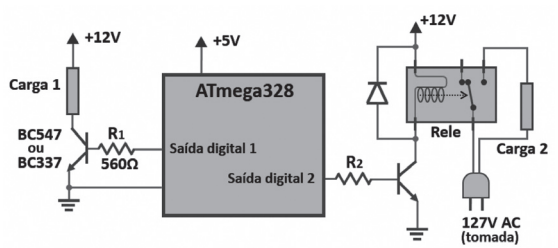
Técnicas de *Debounce*

Podem ser realizadas tanto em hardware quanto em software. A primeira utiliza componentes conectados ao circuito de entrada, formando literalmente um filtro, passivo ou ativo, principalmente para a aquisição de sinais analógicos. Para sinais digitais de entrada, apenas um filtro RC já é suficiente. É verdade que já existe, em cada canal, um resistor interno de *pull-up* e um capacitor, que compõem um filtro. No entanto, por não ser selecionável, e para não atrapalhar em algumas aplicações, a capacitância usada é muito baixa para botões ou chaves (10 pF), e por isso é indicado que se utilize um capacitor externo em paralelo, na ordem de 100 nF. Dessa forma, quando o botão é acionado, as variações, que seriam bruscas, são amortizadas pelo carregamento e descarregamento do capacitor, que gera como resultado uma transição única a cada pressionamento. Outra forma é por software, que seria "menos custosa", pensando em grandes escalas. Esta consiste simplesmente em contabilizar apenas a primeira transição, dentro de uma sequência de transições seguidas (*bounce*). A ideia é que, assim que uma transição for detectada, o sistema passe por um período (preestabelecido) sem contabilizar novos acionamentos, seja por rotinas de *delay*, ou uso de contadores. Para botões em geral, este período pode ser de 100 a 200 ms.

Acionamento de cargas

Como é de se esperar, existem limites energéticos para o funcionamento das portas digitais. No Atmega328, os valores máximos permitidos de corrente são aproximadamente: 40 mA por pino, 100 mA por porta digital e 200 mA para todas as portas do chip. Dessa forma, para acionamento de cargas maiores do que o suportado, é necessário que se utilize alguns componentes externos como interface, como transistores ou relés.

Figura 3.6 | Exemplo de acionamento de dois tipos de cargas digitais



Fonte: elaborada pelo autor.

Na Figura 3.6, repare no diodo conectado em paralelo ao relé. Este é chamado de diodo de "roda livre" ou de *flyback*, e serve para escoar a corrente reversa gerada quando o relé é desligado, devido à carga indutiva armazenada na bobina, e consumida pela própria resistência da bobina. Se não houver esse diodo, os picos de tensão que ocorrem no coletor do diodo acoplado, quando a bobina do relé é desligada, serão tão altos que queimarão o transistor e possivelmente o μC.

Pinos desconectados (não utilizados na aplicação)

Se alguns pinos não forem utilizados pela aplicação, o que é muito comum, é recomendado que estes possuam um nível definido, e não se mantenham flutuantes (em alta impedância, sem nenhum resistor de *pull-up/down*), pois isso pode causar consumo de corrente desnecessário. O método mais simples, e na maioria dos casos mais eficaz, é utilizar o resistor interno de *pull-up* para os pinos não utilizados. Devemos lembrar que é necessário configurar sempre na inicialização os resistores de *pull-up* internos, pois quando o sistema é reiniciado, estes são desabilitados. Se a aplicação possui necessidades críticas de estabilidade, o que é raro, recomenda-se usar resistores externos de pull-up ou pull-down. A conexão dos pinos não utilizados diretamente aos sinais de alimentação VCC ou GND (sem resistor) não é recomendada, pois isso também pode causar correntes excessivas e até danos se o canal for acidentalmente configurado como saída. Um outro motivo para se usar entradas ao invés de saídas para os pinos não utilizados, é que, principalmente na prototipagem, há chance de o desenvolvedor causar alguns "curtos-circuitos" nos pinos do microcontrolador, ao manusear a placa ou protoboard. E, forçar o valor sobre uma entrada não causa nenhum problema, pois, internamente, este sinal é de alta impedância, e existem resistores de *pull-up/down* para limitar a corrente. No entanto, isto não se aplica para as saídas, pois se um canal foi definido como saída em nível alto (VCC) e esta é conectada diretamente ao GND, será drenada muita corrente pelo microcontrolador, que pode ser danificado. Pensando de forma geral, quanto mais saídas existirem no microcontrolador, maiores são as chances de você, acidentalmente, gerar um curto-circuito nos terminais ao manusear e causar algum prejuízo.

Funções alternativas dos pinos

Você deve ter se perguntado em algum momento: se todos os pinos pertencem às portas digitais, como os outros periféricos conseguem acessar o mundo externo? Assim como os demais microcontroladores, a maioria dos pinos são compartilhados entre portas digitais e outros periféricos internos, estrategicamente arranjados.

Pesquise mais

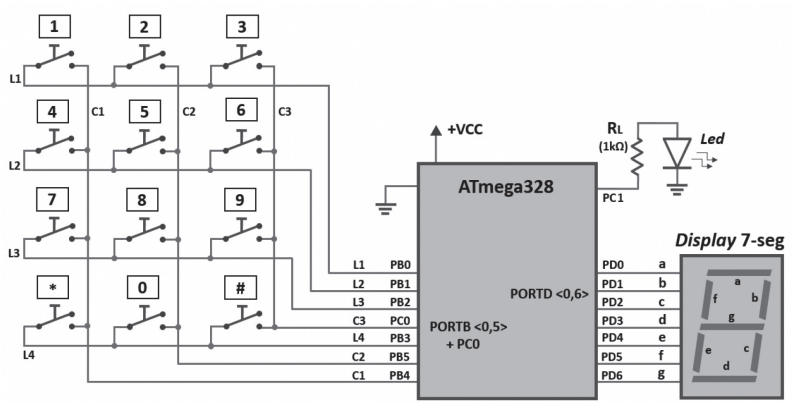
Apesar de não conhecermos os outros periféricos internos do ATmega328 para utilizar, pois ainda vamos estudá-los em detalhes, veja como é feita a configuração para funções alternativas dos pinos em: LIMA C. B. D; VILLAÇA M. V. M. **AVR e Arduino Técnicas de Projeto**. Florianópolis: [s.n.], 2012.

Veja também como montar um sistema básico no protoboard. Disponível em: <http://br-arduino.org/2016/01/arduino-standalone-protoboard.html>. Acesso em: 27 set. 2017.

Sem medo de errar

Vamos agora começar a construir nossos projetos reais, pois já sabemos programar, elaborar algoritmos para soluções de automação, e também transferir o código para um circuito montado com o Arduino UNO (mesmo que simples). Os equipamentos usados para essa tarefa podem ser facilmente encontrados nos laboratórios da sua instituição, ou até simulados, caso não tenha acesso. Estes são: a placa Arduino UNO, um *protoboard* (matriz de contatos), um display de 7 segmentos para visualizar o comando (que pode ser substituído por um *led*), e fios para as devidas conexões. Na tarefa proposta, você deve desenvolver um dispositivo de interface com o usuário da casa, e sua responsabilidade é o teclado matricial, que identificará para o sistema os comandos numéricos. Para que não se utilize muitos pinos para a interface com o teclado numérico, é possível utilizar a técnica de "varredura", em que grupos de teclas compartilham um único sinal de entrada do microcontrolador. Dessa forma, ao invés de se utilizar um pino para cada tecla (4x3=12), podemos arranjar as conexões do teclado em forma matricial, utilizando 4 linhas e 3 colunas (4 + 3 = 7). Esse esquema de conexões pode ser visto na Figura 3.7:

Figura 3.7 | Esquemático proposto como solução do problema (hardware)



Fonte: elaborada pelo autor.

Para permitir o compartilhamento de teclas, é preciso definir estados consecutivos para leitura, onde em cada estado uma das teclas do grupo está "habilitada" para ser verificada, e as demais estão "desabilitadas", ou seja, mesmo que esteja sendo acionada não será reconhecida. Assim, se esses estados forem alternados de maneira "rápida", ou seja, o tempo de acionamento não pode ser menor que um ciclo em todos os estados, é possível o compartilhamento sem consequências indesejadas. Para "habilitar" ou não uma tecla, usaremos sinais de saída, que serão as colunas, por exemplo. Podemos aproveitar os resistores internos de *pull-up*, e, portanto, devemos acionar em nível baixo (as chaves serão todas NF - normalmente fechado). Assim, se todas as linhas forem entradas com *pull-up*, e apenas a coluna 1 está em nível zero, a linha 1 só apresentará zero se a tecla 1 for pressionada, como podemos observar na Figura 3.7. Da mesma forma, se apenas a coluna 2 estiver em zero, apenas as teclas desta coluna poderão ser identificadas pelos sinais de entrada. Vale lembrar que aqui a técnica de *debounce* pode ser feita via hardware, adicionando capacitores em paralelo com o capacitor interno de cada pino. Na verdade, aqui isso se mostra desnecessário, pois o *delay* imposto pelo programa, logo após que uma tecla é pressionada (ou que o primeiro pulso da sequência seja detectado), já funciona como um *debounce* via software, ou seja, os múltiplos pulsos sequenciais gerados pelo pressionar de uma tecla não causarão mais de uma ação. O programa pode ser escrito a partir desse princípio.

Figura 3.8 | Programa proposto como solução para o problema (software)

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>

#define TRUE  1
void Liga_C1(void){
    PORTC |= 0x01;
    PORTB |=0x30; PORTB &=0xEF;}

void Liga_C2(void){
    PORTB |=0x30; PORTB &=0xDF;}

void Liga_C3(void){
    PORTB |=0x30; PORTC &=0xFE;}

unsigned char LeituraTeclado(void){
   unsigned char Tecla= 'n';
        //variável local: n de "nenhuma"
   Liga_C1();  _delay_ms(10);
   if(!(PINB & 0x01)) Tecla = '1';
   if(!(PINB & 0x02)) Tecla = '4';
   if(!(PINB & 0x04)) Tecla = '7';
   if(!(PINB & 0x08)) Tecla = '*';
   Liga_C2();  _delay_ms(10);
   if(!(PINB & 0x01)) Tecla = '2';
   if(!(PINB & 0x02)) Tecla = '5';
   if(!(PINB & 0x04)) Tecla = '7';
   if(!(PINB & 0x08)) Tecla = '0';
   Liga_C3();  _delay_ms(10);
   if(!(PINB & 0x01)) Tecla = '3';
   if(!(PINB & 0x02)) Tecla = '6';
   if(!(PINB & 0x04)) Tecla = '9';
   if(!(PINB & 0x08)) Tecla = '#';

   return Tecla;
}
```

```
const unsigned char Disp7segLookup[12] = {
   0x3f, 0x06, 0x5b, 0x4f, 0x66,  //saídas
   0x6d, 0x7d, 0x07, 0x7f, 0x6f,   //acionadas em 1
   0x77, 0x78    //  'A' e 't'
};

int main(void){       //função principal

   unsigned char Tecla;
   PORTB = 0x3F;   //Saídas em nível 1 e
         //entradas com pull-up interno
    DDRB =  0x30;    //PB0 ao BP3 entradas (linhas)
         //e PB4 e PB5 saídas (colunas 1 e 2)
    PORTC = 0x01;  //desativa coluna 3 e led
    DDRC =  0x03;    //PC0 e PC1 saídas (col3 e led)
    PORTD = 0x00;   //desliga segmentos do display
    DDRD =  0x7F;    //PD0 ao BD6 saídas (disp 7-seg)

    while(TRUE){
       Tecla = LeituraTeclado();  // lê por "varredura"
       if(Tecla!='n'){      //checa se tecla foi apertada
          PORTC |= 0x02;  //liga o led
          if(Tecla=='*'){  //  'A' sterisco
             PORTD = Disp7segLookup[10];
          }else if(Tecla=='#'){    //  't' ralha
             PORTD = Disp7segLookup[11];
          }else{
             Tecla -= 0x030; //converte char para num
             PORTD = Disp7segLookup[Tecla];
          }
          _delay_ms(1000);
          PORTC &= 0b11111101; //desliga o led
          PORTD = 0x00;   //desliga o display
       }
    }
}
```

Fonte: elaborada pelo autor.

Avançando na prática

Câmera fotográfica para pássaros autônoma

Descrição da situação-problema

Suponha que você foi incumbido de desenvolver o projeto de um sistema que faça fotografias automaticamente, a partir do sinal de um sensor de movimento. Esse conjunto deve ser usado para capturar imagens de pássaros silvestres, em que alguns são difíceis de serem encontrados pessoalmente. Além do sensor de presença como entrada, o sistema opera controlando duas saídas digitais: liga a câmera e tira a foto. As especificações técnicas do projeto são as seguintes:

1- Durante um longo período sem a detecção de movimento, a câmera fotográfica deve permanecer desligada (sem alimentação), para poupar energia. Este estado é o HIBERNANDO.

2- Assim que detectado o movimento, a câmera deve ser ligada e aguarda-se um período de 3 segundos para a sua inicialização. Após isso, devem ser tiradas 4 fotos com intervalos de 2 segundos entre elas. A partir deste ponto, o sistema se encontra no estado OPERANDO, onde o sensor de movimento é constantemente monitorado. Durante esse estado, se detectado o movimento, a câmera deve novamente disparar 4 vezes, com intervalos de 2 segundos. O conjunto permanece OPERANDO até que o sensor se mantenha por 30 segundos contínuos sem detectar movimento. Passados 30 segundos sem movimento, a câmera é desligada e o sistema volta ao "HIBERNANDO".

3- A câmera precisa ser acionada por um pulso 100 ms para considerar o sinal de disparo. Portanto, o comando pode ser: "saída alta"; _delay_ms(100); "saída baixa"; _delay_ms(1900);

4- Quando ativo (5 V), o sinal do sensor tem uma duração mínima de 1,5 segundo.

Inicialmente o seu trabalho é escrever um programa para o ATmega328 que atenda às especificações técnicas. Como parte final, e de maneira completa, você deve considerar também o último requisito do sistema: a câmera conta com um *buffer* (armazenador) interno de 6 posições, ou seja, a memória RAM da câmera pode reter até 6 fotografias em uma fila aguardando para serem processadas e armazenadas no cartão SD. Cada foto demanda 5 segundos para ser processada e armazenada. Se o *buffer* estiver cheio, os comandos de disparo são ignorados. Descreva um software que garanta que o sistema não execute um comando de disparo em vão, mas que aproveite com eficiência o *buffer* circular (dica: monte um diagrama temporal de uma realização para ver o comportamento do seu algoritmo, e use uma variável para armazenar quanto tempo, em segundos, falta para o *buffer* ficar vazio, com um *tick* de 1 segundo).

Resolução da situação-problema

Usando e adequando os exemplos sobre sincronismo de tarefas através de contadores que estudamos, podemos resolver aqui os controles temporais do nosso problema.

Figura 3.9 | Programa proposto como solução para a parte b) do problema (geral e completo)

```
#define F_CPU 16000000UL

#include <avr/io.h>
#include <util/delay.h>

#define TRUE  1

#define esp  _delay_ms

#define SENSORP (1 << PB0)
#define DISPARA  (1 << PC0)
#define HABCAM  (1 << PC1)

void TiraFoto(void){
    PORTC |= DISPARA;   esp(100);
    PORTC &= (~DISPARA);   esp(1900);
}

int main(void){

    unsigned int i, j, Cnt_buf;

    PORTB = 0xFF;     //entrada com pull-up
    DDRB = 0x00;      //todos como entradas
    PORTC = 0x00;     //desativa saídas
    DDRC = (DISPARA | | HABCAM); //config

    while(TRUE){       //Loop infinito

        if(!(PINB & SENSORP)){  //se detectar presença
            //em HIBERNANDO
            PORTC |= HABCAM;    //liga a câmera
            esp(3000);  //aguarda a inicialização da câmera

            for(i=0;i<4;i++)      //tira 4 fotos
            TiraFoto();
            //4 fotos levam 20seg para serem processadas
            //no entanto, cada foto leva 2 seg para ser tirada
            Cnt_buf=12;  //agora faltam 12 seg
            //para limpar o buffer
            for(i=0; i<30; i++){  //entra no time-out (OPERANDO)
                if(!(PINB & SENSORP)&&(Cnt_buf<26)){
                    //se tem presença e tem vaga no buffer
                    for(j=0; j<4&&(Cnt_buf<26); j++){ // tira 4 fotos ->
                        TiraFoto();    //passou 2 seg (process da foto)
                        Cnt_buf+=3; //e uma foto foi adicionada (5-2=3)
                    }
                    i=0;          //-> e reseta a contagem de 30 seg
                    }else{  //se não detectar presença em OPERANDO
                    esp(1000);          //fatia do time-out: 1 seg  (tick)
                    if(Cnt_buf) Cnt_buf--;  //se tiver,
                }                              //foi processado 1 seg da foto
            }
            PORTC &= (~HABCAM);    //se passou o time-out
        }            //desliga a câmera e volta para HIBERNANDO
    }
}
```

Fonte: elaborada pelo autor.

Faça valer a pena

1. Muitos dos projetos de sistemas embarcados utilizam as portas digitais do microcontrolador. Isso porque grande parte dos sinais utilizados nas aplicações são digitais, apesar de termos aquela velha ideia de que o mundo é contínuo. Apesar de, teoricamente, alguns sinais serem de fato contínuos, como os populares sinais de tempo ou de espaço, alguns outros são discretos ou digitais por natureza.

A respeito das portas digitais presentes no microcontrolador ATmega328, considere as seguintes afirmações:

I – Os canais podem ser configurados como saída e como entrada simultaneamente, atuando de maneira "híbrida" quando o pino está ligado a sinais "fracos" externos (baixa energia).

II – As portas digitais no Atmega328 possuem um registrador usado exclusivamente para habilitar os resistores de *pull-down* internos, mas apenas nas portas A e B.

III – Os resistores internos que podem ser habilitados nas portas podem ser utilizados tanto em entradas quanto em saídas, seguindo a coerência lógica da aplicação em questão.

IV – Cada canal digital pode estar em um dos três estados possíveis (entrada, saída alta, saída baixa), o qual é controlado pelo programa usuário, e pode ser modificado entre seus estados livremente.

Considerando a ordem apresentada, qual das alternativas representa corretamente as afirmativas verdadeiras e falsas?
a) F, V, F, F.
b) F, F, V, V.
c) V, F, F, V.
d) F, F, F, F.
e) F, F, V, F.

2. Nos sistemas digitais ou sistemas embarcados, os sinais geralmente possuem um sentido de corrente bem definido. No entanto, o desenvolvedor deve se atentar se todas as conexões apresentam coerência elétrica, pois duas saídas não podem compartilhar diretamente um mesmo barramento, por exemplo. Inclusive os pinos que não são utilizados em uma determinada aplicação devem ser considerados.
O que ocorre, ou como é o comportamento do sinal elétrico de um condutor metálico que não esteja conectado a nenhum outro circuito, ou seja, em alta impedância?
a) Não é possível afirmar nada sobre esse valor, muito menos prevê-lo, pois este será, teoricamente, igual ao potencial elétrico aplicado ao condutor na última vez, que se mantém constante se a corrente não pode ser "descarregada" para outro lugar.
b) Não é possível afirmar qual é esse valor, no entanto, podemos dizer que este sinal não é constante, e se mantém "flutuando" de acordo com as ondas eletromagnéticas presentes no ambiente. Esse sinal pode inclusive assumir altos valores, uma vez que é um sinal apenas de tensão, e não de potência (não há corrente).
c) Não é possível afirmar qual é o seu valor, pois este pode ser representado por um sinal aleatório que se mantém em uma única frequência, e varia de acordo com um ruído branco (sinal aleatório com todas as frequências de uma banda muito larga). Esta variação depende unicamente das estruturas químicas do material, que apresenta esse mesmo padrão mantido durante sua existência, independentemente do meio em que está.
d) Podemos afirmar que o sinal do potencial elétrico será zero, ou muito próximo, considerando que o condutor foi deixado nesse estado por um "longo" período de tempo. Sempre que um condutor metálico, independente do campo magnético, é isolado de qualquer circuito ou potencial, sua carga elétrica inicia o processo de dissipação no meio onde está, e vai se descarregando, muito lentamente, assim como um capacitor

carregando. Se verificado após um longo tempo, o potencial encontrado será apenas um "resquício" da última carga imposta no condutor.
e) Não é possível prever o sinal elétrico, apenas se pode afirmar que, através da relação entre campos elétricos e magnéticos, o sinal sofre fortes influências do campo magnético terrestre, e portanto, apresenta variações para lugares diferentes no globo. É esperado, portanto, que um condutor elétrico isolado, que não seja transportado geograficamente, mantenha seu potencial elétrico estático (se não houver nenhuma fuga/chegada de corrente ou carga).

3. As portas digitais presentes em praticamente todos os microcontroladores atuais possuem poucos comandos para se manusear, bem como poucos registradores para configurar, quando comparadas com outros periféricos internos comuns. Por exemplo, para uma entrada, o único comando utilizado é a leitura (além da configuração). Uma saída pode ser acionada em nível alto ou baixo. A dificuldade encontrada no desenvolvimento de programas que utilizam portas digitais é o "momento em que essas simples ações devem ser feitas".
Considerando a manipulação das portas digitais do microcontrolador ATmega328, em linguagem de programação C, qual das alternativas **não** pode ser utilizada para inverter o estado de um *led* conectado à Porta B, canal zero (já pré-configurado)?
a) PORTB = PORTB ^ 0xFF;.
b) if(PINB & 0x01) PORTB = 0x00;
else PORTB = 0x01;.
c) PINB ^= 1;.
d) PORTB = (PINB & 0x01)? 0:1;.
e) if(PORTB && 0x01) PORTB &= 0x01;
else PORTB |= 0x01;.

Seção 3.2

Interrupções

Diálogo aberto

Como os demais, esse assunto é muito importante e carrega suas particularidades de funcionamento e aplicações. Aqui vamos ver como as interrupções ocorrem e quais são as implicações em hardware e em software. Veremos que, depois de configurada e habilitada, uma fonte de interrupção pode imediatamente desviar o programa para um trecho pré-definido, para tratar em tempo real o evento ocorrido. Esse mecanismo é muito importante para a criação de sistemas de baixo consumo, que podem se manter inativos quando não precisam atuar, mas sem comprometer a latência de tratamento das ações. Além de alertar o núcleo que uma interrupção ocorreu, as interrupções servem também para "despertar" o núcleo, ou outras partes do sistema que estejam "hibernando", ou seja, temporariamente inativas. Depois de entender como funciona o vetor de interrupções interno, e como as interrupções externas devem ser configuradas para determinadas aplicações, você deve resolver uma questão com esse assunto para entender melhor ainda como deve atuar em novas soluções. Nessa situação, você trabalha em equipe para desenvolver um dispositivo de interface com o usuário de uma casa automatizada. Cada um da equipe é responsável por uma parte do projeto de automação doméstica, e a sua tarefa foi desenvolver a interface, tanto em hardware quanto em software, para o teclado matricial, que identificará para o sistema os comandos numéricos. Portanto, você fará agora sistema "teste", que é autossuficiente, para que depois possa ser incluído no protótipo inicial. Sabendo que utilizaremos um teclado numérico matricial para esse projeto, algumas decisões devem ser tomadas para realizar a conexão entre os sinais das teclas e o microcontrolador Atmega328.

O programa final será composto por diversas rotinas para controlar as saídas de forma eficiente e sincronizada. Inicialmente, você desenvolverá algumas dessas rotinas básicas, aplicadas para um sistema equivalente, porém, mais simples, que serão usadas

para o protótipo final. Refaça o mesmo projeto da situação-problema anterior, mas com uma tecnologia diferente, usando agora interrupção externa, além de algumas funcionalidades adicionais. Isso permitirá que o programa não fique checando incessantemente as entradas, e permaneça em modo "*sleep*", ou "hibernação", economizando consumo de energia. Nesse novo cenário, o display deve apresentar dois valores para cada acionamento de botão, com um segundo de duração cada. O primeiro valor representa a tecla pressionada, como no caso anterior, e o *Led* deve permanecer aceso juntamente. Logo após, deve ser exibido a quantidade de vezes que aquela tecla foi apertada desde a última inicialização do conjunto, enquanto o *led* está apagado, e, depois do intervalo, tudo se apaga. Diferentemente do problema anterior, agora o sistema deve detectar um acionamento enquanto exibe um outro anterior, dessa forma, se no início a tecla 5, por exemplo, for pressionada duas vezes rapidamente, o segundo valor que deve aparecer é o 2, omitindo a exibição para o primeiro acionamento. Por usar interrupção, o sistema pode não fazer nada no loop principal (apenas nas rotinas de interrupção), ou realizar as devidas tratativas através do uso "*flags*", acionadas na interrupção. Para ambos os casos, o tratamento de "*debounce*" pode ser feito via software? Se em caso positivo, como? Esse esforço computacional pode ser evitado com um tratamento via hardware? Se sim, como?

Não pode faltar

Estamos entrando em um novo patamar para a criação de sistemas embarcados ou computacionais: fornecer ao sistema a capacidade de trabalhar apenas nos momentos necessários, permanecer semidesligado (hibernando) no resto do tempo para poupar energia, sem comprometer sua performance. Deve ficar claro que a única energia economizada é a que seria consumida pelo chip. Portanto, não podemos afirmar que o "sistema está em modo de baixo consumo" se o núcleo entra em modo sleep (dormindo), mas periféricos internos e externos se mantêm em atividade, como um *led* ou um motor, tornando a economia do núcleo desprezível. É interessante que o sistema desative todos os elementos consumidores de corrente possíveis quando passa

por longos períodos sem atividade. Vamos ver agora um outro tipo de classificação usado em sistemas computacionais, para o tratamento de tarefas, feito individualmente. São o "*polling*" (minar) e a interrupção. Dessa forma, como já retratamos muitas vezes, um sistema grande ou complexo pode ser dividido em tarefas menores, que devem controlar uma saída (ou um grupo que opere junto) ou um processo específico, que até então era feito apenas através de funções. Agora, você será apresentado a um novo método de tratamento/controle, que são as interrupções. Antes de tudo, devemos deixar claro o conceito de interrupção. Uma interrupção, em sistemas computacionais, é um sinal digital interno que (se habilitado) indica à CPU que uma interrupção ocorreu e que, portanto, uma rotina pré-configurada e associada a esse sinal de interrupção deve ser executada, de acordo com sua prioridade, arbitrada pelo Centro de Controle de Interrupções. Em praticamente todos os sistemas computacionais existe o emprego de interrupções, e cada uma delas é reservada para um dos periféricos internos. Como ainda não estudamos os periféricos, com exceção das portas digitais, não faz sentido apresentar agora todas as fontes de interrupção do ATmega328, e serão estudadas no devido momento.

Polling x Interrupção

Uma confusão que ocorre muitas vezes quando esse tema é apresentado é a separação entre funções e rotinas de tratamento de interrupção (ou apenas rotinas de interrupção). A função possui de fato um papel semelhante ao de uma rotina, que é desviar o programa para tratar um determinado assunto, mas pode estar presente nas interrupções, e podem ser usadas ou não para o método de *polling*, independentemente. A diferença essencial é que a rotina é um trecho de programa que é executado sem a invocação por outra função, com a única exigência de estar devidamente configurada e habilitada. Já as funções são trechos de programa que são invocados pela função principal, ou outra qualquer. Ou seja, o programa usuário "sabe" quando o trecho vai ocorrer.

Exemplificando

Vamos considerar a situação em que você possui duas tarefas a serem feitas: manter a conversa que está tendo com um colega de trabalho, e responder um e-mail importante que chegará em breve. Sabendo que a segunda tarefa tem uma prioridade muito maior, mas que você não quer simplesmente "parar" a tarefa 1 para ficar aguardando o e-mail, é possível escolher entre duas formas para atender ao tratamento da tarefa 2: a primeira é "***polling***", que corresponde a você, enquanto conversa, checar o e-mail no celular a cada um minuto, por exemplo. Dessa forma, a tarefa 1 não é comprometida, e nem a 2; a segunda maneira, que geralmente é mais conveniente, é por "**interrupção**": ao invés de se manter checando o celular toda vez, é possível configurar o seu aplicativo de e-mail para acionar um sinal sonoro de aviso quando receber um e-mail novo. Além de economizar a "energia" de ficar checando a condição para tratar a tarefa, ela é atendida muito mais rapidamente nesse segundo método.

Dessa forma, podemos deduzir que a separação das saídas (ou tarefas) que serão tratadas por *polling* ou interrupção não é feita da mesma forma com que separamos as tarefas do programa em funções. É verdade que muitos profissionais afirmam que, nos sistemas computacionais que se transformam em produtos finais, todos os tratamentos devem ser feitos via interrupção. No entanto, o uso do polling se mostra útil em algumas ocasiões.

Pesquise mais

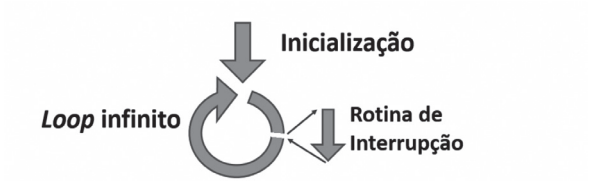
Procure saber mais detalhes sobre as técnicas de *polling* e interrupção. Disponível em: <http://www.din.uem.br/sica/material/8259/8259.html>. Acesso em: 27 set. 2017.

Podemos afirmar que as interrupções são como as funções, porém, não são invocadas por nenhuma outra, e não possuem parâmetros nem de entrada nem de saída. O que cada fonte de interrupção permite como controle é a sua configuração (modo de operação) e sua habilitação, que permite que a interrupção ocorra quando o sistema encontrar as condições preparadas pelo programador. Dessa forma, não podemos classificar as rotinas de tratamento de interrupções na estrutura de camadas que aparece

quando o programa é dividido em funções, como estudamos. No entanto, é comum a classificação dessas rotinas como de camada mais baixa em software, chamada de HAL – *Hardware Abstract Layer*, ou camada de abstração de hardware. Devemos estar atentos para não confundir o sentido de classificação das camadas. Em seções anteriores, foi visto que a *main* é a primeira camada, e assim que as funções eram invocadas, a ordem ia aumentando. De fato, diz-se o contrário, ou seja, a função da "ponta" de uma cadeia de chamadas sucessivas é a que está na camada mais baixa, geralmente em contato direto com o hardware (registradores e rotinas de interrupção). Uma outra maneira de se classificar as interrupções de forma hierárquica é através da prioridade associada a cada uma delas, que são inalteráveis. Assim, se ocorrerem duas interrupções juntas, ou se ocorre uma interrupção enquanto outra não foi completamente atendida, a interrupção de maior prioridade assume o controle do processo, e as outras aguardam a sua vez, como será visto em "interrupções aninhadas".

Podemos observar que o momento em que as interrupções ocorrem não pode ser previsto pelo programa principal, e ocorrem em instantes aleatórios, interrompendo o programa onde quer que esteja. Podemos exemplificar a estrutura de um programa com uma única interrupção na Figura 3.10. Deve-se ressaltar que a figura retrata uma realização de um único ciclo, interrompido. Outros ciclos podem não ser interrompidos, como também podem ser, em qualquer instante.

Figura 3.10 | Estrutura básica de um programa embarcado, para uma realização do ciclo



Fonte: elaborada pelo autor.

Modos de hibernação

Apesar de não nos aprofundarmos muito, vamos aprender como funcionam os modos de hibernação, também conhecidos como *sleep modes*, e como se relacionam com as interrupções.

Pensando em termos práticos, é recomendado que um sistema possa funcionar plenamente com o mínimo consumo de energia possível, principalmente em aparelhos com baterias. Se pararmos para observar, a maioria dos aparelhos eletrônicos passam por longos períodos de inatividade, mas que não devem ser desligados, ficando prontos para operar quando necessário. Vamos lembrar do nosso exemplo do controle automático do portão de garagem. O sistema passa grande parte do tempo ocioso, esperando o proprietário acionar o botão, para então acionar os motores. Nesse período, o sistema não precisa estar em atividade, e pode entrar em modo "*sleep*". No entanto, antes disso, o programa deve configurar uma interrupção para ocorrer, caso o sinal do botão seja recebido, o que "acordaria" o processador para trabalhar. Quando isso ocorre, as saídas são acionadas, e a próxima tarefa é desligar o motor quando o sensor fim de curso for atingido, e essa tarefa pode ser tratada de maneira independente da primeira, o que deve ficar claro para você, aluno, pois, ainda que a tarefa do botão tenha sido tratada por interrupção, essa próxima pode ser por *polling*, ou seja, o processador pode continuar ativo, consumindo mais energia apenas enquanto o portão se move, e se manter checando o estado da entrada do sensor. Por outro lado, o programa usuário poderia configurar uma interrupção para ocorrer quando o sensor for acionado, e na rotina de tratamento desta interrupção, desligar o motor.

Geralmente, os aparelhos eletrônicos são feitos para consumir menos energia e, para isso, utilizam as interrupções e os modos de hibernação, que podem ser muitos, inclusive. Nesse caso, cada modo corresponde a um conjunto de periféricos internos que ficam em atividade. O programa usuário pode encerrar a execução do programa (atividade no núcleo) e manter o periférico de comunicação em atividade. O seu celular, por exemplo, desliga alguns periféricos não essenciais quando fica inativo, como a tela. O Atmega328 possui 6 diferentes modos de hibernação.

Pesquise mais

Não é essencial para desenvolver soluções, mas é muito importante saber como operam os modos de hibernação do Atmega328, principalmente

para o desenvolvimento de projetos que serão utilizados por muito tempo. Veja as características principais no capítulo 14: PM - *Power Management and Sleep Modes*, na página 62 do manual (*datasheet*) oficial do ATmega328. Disponível em: <http://www.atmel.com/pt/br/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso em: 27 set. 2017.

Apenas o chip do ATmega328, que consome 0,2 mA em modo ativo, chega a consumir 100 nA no modo mais econômico ("*power down mode*"). Associando à sua tensão de trabalho mínima, 1,8 V, o chip pode consumir no mínimo 180 nW de potência.

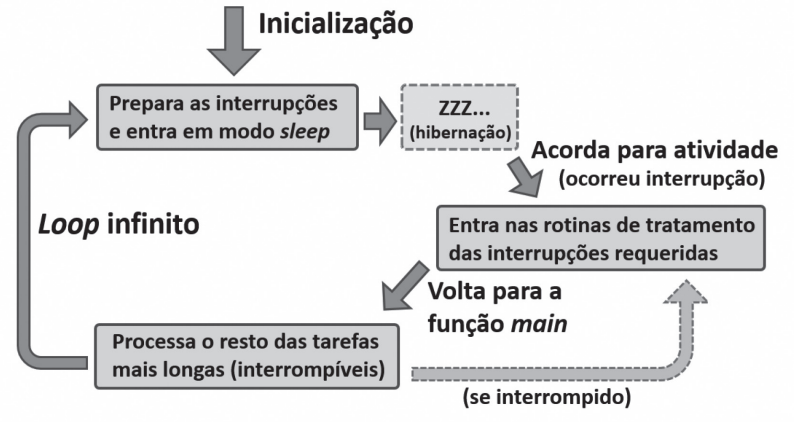
Assimile

Apesar de não ser tão importante para estudo e prototipagem, o uso dos modos de hibernação é muito importante para aplicações que funcionarão por muito tempo. Dessa forma, quando for elaborar um projeto que será utilizado por alguém, tente adaptá-lo para operar em hibernação sempre que possível (depois de terminar por completo em modo ativo).

Funções de rotinas de tratamento

Apesar de não ser obrigatório, é altamente recomendado que dentro das rotinas de interrupção, se escreva o mínimo de código possível, e que não se invoque funções. Isso serve para não causar muitos aninhamentos (sobreposições) de rotinas, quanto também de funções, que é fonte comum de problemas imprevisíveis. Dessa forma, se o tratamento da tarefa é simples e rápido, como apenas transmitir um dado, ou acionar uma saída, por exemplo, deve ser feito na própria rotina, sem uso de funções. Caso o tratamento seja mais complexo ou demorado, como imprimir uma imagem em um display gráfico, este deve ser feito na função principal, que pode utilizar outras funções, e pode ser interrompida por outras fontes com menos risco. Podemos visualizar assim uma estrutura de programa clássica para sistemas embarcados com economia de energia. Ela é parecida com a anterior, com a diferença do núcleo "dormir" sempre que possível.

Figura 3.11 | Estrutura básica de um programa embarcado com interrupções e hibernação



Fonte: elaborada pelo autor.

Interrupções no ATmega328

O programa que vai embarcado é de fato iniciado na função *main*; no entanto, as primeiras instruções da função *main* não estão nos primeiros endereços de memória de programa. Esta região é reservada como vetor de interrupções do sistema, e os seus valores (em cada par de endereços) são interpretados como "ponteiros" (de fato, instruções JMP XX), para onde o programa deve ser desviado na memória de programa, ou seja, são os desvios para as rotinas de tratamento de cada uma das interrupções. Nesse caso, inclusive a função *main* pode ser considerada como uma "rotina de interrupção", pois é invocada pelo vetor de interrupções, no seu primeiro endereço (zero), o qual corresponde à interrupção principal: RESET. Dessa forma, sempre que o chip é inicializado, ou resetado, o ponteiro de programa PC aponta para o endereço zero da memória de programa (controlado pela unidade de controle e interrupções), e salta para a região da memória onde as instruções da função principal estão realmente alocadas.

Pesquise mais

Os demais primeiros endereços possuem desvios para as rotinas de interrupções específicas, que são identificadas pelo programador por

palavras-chave já definidas pelo fabricante. A interrupção externa para as portas digitais tem sua rotina indicada no programa em C pelo rótulo INT0, como veremos nos próximos exemplos. Os demais rótulos podem ser vistos na tabela 16-1 do manual do fabricante: "*Table 16-1. Reset and Interrupt Vectors in ATmega328/P*" presente na página 82. Disponível em: <http://www.atmel.com/pt/br/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso em: 27 set. 2017. Perceba que as prioridades das interrupções desse modelo não podem ser alteradas, e seguem a ordem deste vetor (menor endereço, maior prioridade).

Cada interrupção do vetor de interrupções pode ser configurada independentemente, e, portanto, possuirá uma rotina individual para ser executada caso a interrupção ocorra. No entanto, devemos perceber que as portas digitais possuem uma interrupção para cada uma, mas todos os canais de uma mesma porta compartilham uma única fonte de interrupção. Dessa forma, se uma aplicação demandar duas interrupções, e permite utilizar dois canais digitais de portas distintas, será possível atender a cada uma das interrupções de maneira exclusiva. Caso contrário, é preciso que uma mesma porta atenda a mais de uma interrupção, sendo necessário checar qual das entradas gerou uma rotina de interrupção. Isso é feito pelo programa, no início da rotina de tratamento de interrupção, que verifica qual tratamento deve ser realizado por esse acionamento, ou seja, um "*polling*" dentro de uma interrupção.

Interrupções aninhadas

As interrupções podem ocorrer e ser atendidas "simultaneamente", assim como as funções. De fato, apenas uma interrupção é executada de cada vez, pois há um processador, mas uma rotina pode se manter pendente, ou inacabada, aguardando que outra(s), com prioridade maior, termine seu trabalho e libere o processador. Isso ocorre quando uma função invoca outra, e retoma o processamento quando a função chamada retorna. Deve-se estar muito atento para todas as situações de sobreposição das interrupções habilitadas para a aplicação, pois áreas da memória ou intervalos de tempo/

sincronismo podem ser comprometidos com aninhamentos não premeditados, que podem gerar incoerências e anomalias. Como foi dito, a prioridade segue a ordem do vetor de interrupções, com grau de prioridade inversamente proporcional ao endereço.

Interrupção externa

Agora que sabemos o que é interrupção, vamos aprender a primeira fonte de interrupção, que é a interrupção externa, e está associada ao único periférico interno visto, as portas digitais.

Reflita

Para as portas digitais, as interrupções podem ser aplicadas para apenas entradas, ou para entradas e saídas? Pense em possíveis situações para responder a pergunta.

Podemos perceber que as interrupções devem ser usadas apenas nas entradas, pois não faz sentido a transição de uma saída, que é controlada pelo sistema, gerar uma interrupção. As interrupções servem para refletir que eventos da natureza externa ocorreram, e quando foi, para que o programa usuário possa tratar a tarefa apropriada. No entanto, se estas forem habilitadas, podem ser acionadas por canais de saída. Isso pode ser usado para gerar uma interrupção internamente, via software, que é a mesma coisa que invocar uma função. Existem dois tipos de interrupção geradas através dos pinos digitais, para o microcontrolador estudado: INT e PCINT.

INT - Recebem o nome de "interrupções externas" apesar da outra ser também uma interrupção externa. Existem apenas dois canais no chip, o INT0 e INT1, nos pinos PD2 e PD3, como pudemos identificar na Figura 3.1. Através dos bits ISC00/1 - *Interrupt Sense Control 0*, ou controle de sensibilidade de interrupção zero, o canal zero pode ser configurado para gerar interrupção por nível baixo (00), qualquer transição (01), borda de descida (10), ou borda de subida (11). Igualmente, o par de bits **ISC10 e ISC11** configuram o canal 1. Estes dois pares de bits estão, em ordem crescente, nas primeiras posições do registrador **EICRA** - *External Interrupt Control Register A*, ou registrador de controle da interrupção externa A. As fontes de

interrupção podem ser habilitadas (nível 1), ou mascaradas através dos dois primeiros bits (que recebem o mesmo nome da interrupção) **INT0 e INT1** do registrador **EIMSK** - *External Interrupt Mask Register*, ou registrador de mascaramento de interrupção externa. Quando uma interrupção externa ocorre, o bit correspondente, **INTF0/1** - *External Interrupt Flag 0/1*, do registrador **EIFR** - *External Interrupt Flags Register* é acionado. Além disso, se a interrupção externa estiver habilitada, assim como o bit que habilita globalmente as interrupções GIE, no registrador de status, o programa é desviado para a devida rotina de tratamento. Devemos ressaltar que, para estas duas fontes, as transições são detectadas por sinais físicos, via hardware, e, portanto, de maneira assíncrona com o sistema geral. Isso significa que estes sinais de interrupção podem ser utilizados para "acordar" o núcleo que está em modo hibernação, mas só podem ser acionados por nível baixo (transições não servem).

PCINT - *Pin Change Interrupt*, ou interrupção por mudança no pino. Esse tipo de interrupção está disponibilizado em todos os canais de entrada e saída do ATmega328, no entanto, estas possuem algumas limitações em comparação com as duas fontes de interrupção externa INT: uma única fonte de interrupção é compartilhada entre todos os canais de uma mesma porta, e a única maneira de se acionar uma interrupção é por qualquer transição (não configurável). Como pudemos ver na Figura 3.1, a porta B corresponde ao PCINT[7:0], e este grupo compartilha o sinal de interrupção chamado PCI0 - *Pin Change Interrupt Request 0*, ou solicitação de interrupção por alteração do valor de pino 0. Da mesma forma, as portas C e D pertencem aos 16 próximos bits PCINT[15:8] e PCINT[23:16]) e aos sinais PCI1 e PCI2, respectivamente. Apesar de todos os pinos de uma mesma porta serem capazes de causar uma interrupção, é possível habilitar cada canal individualmente, do PCINT0 ao PCINT23, através dos três registradores **PCMSK0/1/2** - *Pin Change Masking Register 0/1/2*, ou registradores de mascaramento de transição do pino. Dessa forma, a porta B pode ter apenas o primeiro e o último canais habilitados para gerar interrupção, através de **PCMSK0 = 0x81;**. Além deste mascaramento, e analogamente ao INT, as três fontes de interrupção PCI0/1/2 podem ser habilitadas pelos três primeiros bits do registrador **PCICR** - *Pin Change Interrupt Control Register*, ou registrador de controle de interrupção por mudança de pino, através dos três bits **PCIE0/1/2** - *Pin Change Interrupt Enable 0/1/2*.

Perceba que existem dois bits para habilitar uma interrupção PCINT. Por exemplo, a PCINT16, localizada no canal PD0, e, portanto, ao terceiro canal de interrupção PCINT (PCI2), deve ser habilitada com **PCMSK2 = 0x01**; (PCINT16) e **PCICR = 0x04;** (PCIE2).

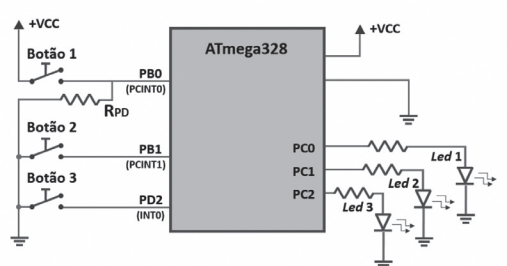
Funções de interrupção: além das interrupções locais serem acionadas, as interrupções também devem ser habilitadas globalmente, através do bit GIE - *General Interrupt Enable*, no registrador de status. Isso é feito através do comando/função **sei()**; (*Set Interrupt Bit*). Esse bit global pode ser apagado através da função **cli()**; (*Clear Interrupt Bit*). Para utilizar essas funções, bem como qualquer interrupção no programa, a biblioteca de interrupções deve ser incluída, como veremos.

Exemplificando

Para esclarecer, vamos tratar do seguinte problema: construir um projeto com três entradas digitais: botões ligados em PB0, PB1 e PD2 (PCINT0/1 e INT0), que controlam igualmente três saídas: *Leds* 1, 2 e 3, ligados nos canais digitais PC0, PC1 e PC2. Com o acionamento do botão, o *led* correspondente deve piscar três vezes em 2,5 Hz (T=400 ms), independente dos demais. Se não houver trabalho, o sistema deve entrar em hibernação.

Para resolver essa questão, vamos utilizar ambas as técnicas de interrupção e *polling*, uma vez que não sabemos ainda gerar referência temporal com interrupções, através dos temporizadores. Dessa forma, o conjunto entra em *sleep*, e acorda se for interrompido (apenas pelo botão 3). A partir daí se mantém piscando o *led* requerido através de contadores, e verifica os momentos de piscar e de fim de trabalho (*polling*). Uma vez que todas as tarefas foram tratadas, mesmo as que apareceram depois, o sistema volta a dormir.

Figura 3.12 | Esquemático para o exemplo



Fonte: elaborada pelo autor.

Figura 3.13 | Programa embarcado para o exemplo

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/power.h>
#include <avr/sleep.h>

#define F_CPU 16000000UL
#include "util/delay.h"
#define TRUE 1

unsigned char FLAG_T1, FLAG_T2=6, FLAG_T3;
            //vars globais
void ConfigINT0(void){
   DDRD = 0x00;          // PD2 entrada
   PORTD |= (1 << PORTD2); //hab pull-up  PD2
   EICRA = 0x00; // config INT0 acionada por nível baixo
                 // (única maneira de sair do modo sleep)
   EIMSK |= (1 << INT0);      // Habilita a interrupção
}
void ConfigPCINT0(void){
   PORTB |= (1 << PORTB1);  //hab pull-up em  PB1
   PCICR |= (1 << PCIE0);   // habilita interrupção na porta B
   PCMSK0 |= ((1<<PCINT0)|(1<<PCINT1)); //hab int canais 0/1
}
int main(void){
   unsigned int i, Cnt1=20, Cnt2=20, Cnt3=20;
   unsigned char FlagTRAB;
   DDRB = (1 << DDB5);    //PB0 e PB1: PCINT0/1;  e PB5: LED
   DDRC = 0x07;  //PC0, PC1 e PC2 saídas: Led1, Led2 e Led3

   ConfigPCINT0();
   ConfigINT0();
   for(i=0; i<10; i++, PORTB^=(1<<PB5)) _delay_ms(50); //Olá

   sei();    // Habilita interrupções (Global) → GIE no status reg
   set_sleep_mode(SLEEP_MODE_PWR_DOWN);//modo usado
   while(TRUE){
      FlagTRAB = FLAG_T1 | FLAG_T2 | FLAG_T3;
      while(FlagTRAB){ //cond p manter "acordado"
         if(FLAG_T1){    // trata tarefa 1
            if(Cnt1) Cnt1--;
            else{ Cnt1=20;  FLAG_T1--;
               PORTC^=(1<<PC0);
            }}
         if(FLAG_T2){    // trata tarefa 2
            if(Cnt2) Cnt2--;
            else{ Cnt2=20;  FLAG_T2--;
               PORTC^=(1<<PC1);
            }}
         if(FLAG_T3){    // trata tarefa 3
            if(Cnt3) Cnt3--;
            else{ Cnt3=20;   FLAG_T3--;
               PORTC^=(1<<PC2);
            }}
         _delay_ms(10);
         cli(); //desabilita globalmente interrupções
         FlagTRAB = FLAG_T1 | FLAG_T2 | FLAG_T3;
         sei();    //Habilita interrupções globalmente
      }
      sleep_enable();
      sleep_mode();  //ou sleep_cpu(); → Vai dormir
      //ZZZ...
      sleep_disable();  //acorda e vai tratar as rotinas
   }
}
ISR (PCINT0_vect){   // rotina de tratam int. PCINT0
   if(PCIFR & (1 << PCIF0)) FLAG_T1=6;
   if(PCIFR & (1 << PCIF1)) FLAG_T2=6;
}
ISR(INT0_vect){ //tratamento da interrupção INT0
   FLAG_T3 = 6;
}
```

Fonte: elaborada pelo autor.

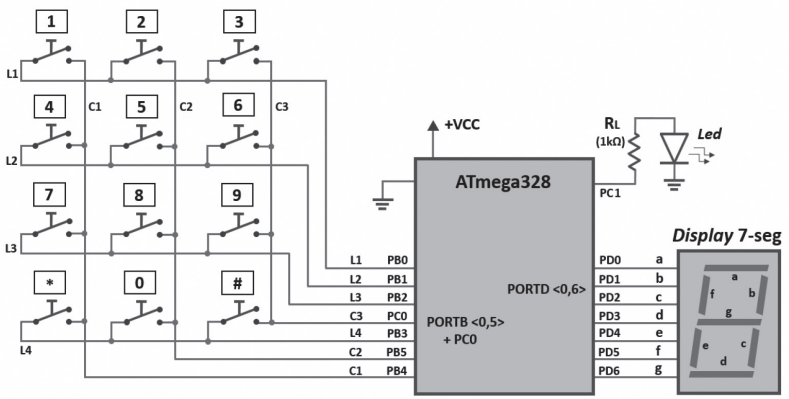
Sem medo de errar

Vamos agora resolver o problema proposto, começando com uma análise sobre suas especificações técnicas. Antes de tudo, vamos relembrar o que deve ser feito. Nessa situação, você trabalha em equipe para desenvolver um dispositivo de interface com o usuário de uma casa automatizada. Cada um da equipe é responsável por uma parte do projeto de automação doméstica, e a sua tarefa foi desenvolver a interface, tanto em hardware quanto em software, para o teclado matricial, que identificará para o sistema os comandos numéricos. Portanto, você fará agora sistema "teste", que é autossuficiente, para que depois possa ser incluído no protótipo inicial. Sabendo que utilizaremos um teclado numérico matricial para esse projeto, algumas decisões devem ser tomadas para realizar a conexão entre os sinais

das teclas e o microcontrolador Atmega328. Nesse novo cenário, o display deve apresentar dois valores para cada acionamento de botão, com um segundo de duração cada. O primeiro valor representa a tecla pressionada, como no caso anterior, e o *Led* deve permanecer aceso juntamente. Logo após, deve ser exibida a quantidade de vezes que aquela tecla foi apertada desde a última inicialização do conjunto, com o *led* apagado, e, depois do intervalo, tudo se apaga. Acontece que agora o sistema deve detectar um acionamento enquanto exibe um outro anterior. Desta forma, se no início a tecla 5, por exemplo, for pressionada duas vezes rapidamente, o segundo valor que deve aparecer é o 2, omitindo a exibição para o primeiro acionamento. Por usar interrupção, o sistema pode não fazer nada no loop principal (apenas nas rotinas de interrupção), ou realizar as devidas tratativas através do uso "*flags*", acionadas na interrupção. Para ambos os casos, o tratamento de "*debounce*" pode ser feito via software? Esse esforço computacional pode ser evitado com um tratamento via hardware?

Um primeiro ponto pode ser a ideia de usarmos um conjunto de entradas compartilhadas, lidas por método de varredura, e devemos incorporar hibernação e interrupções. Se fosse usado um canal para cada tecla, seria muito mais fácil implementar o software, pois cada interrupção seria associada a uma tecla, simples assim. Como uma mesma entrada digital é compartilhada, fica difícil para o programa determinar qual das teclas que causou a interrupção. Uma solução parcial para isso é usarmos *polling* depois da interrupção. Dessa forma, podemos, no loop infinito, deixar todas as colunas (saídas) em nível zero, habilitar as interrupções nas entradas, e entrar em modo *sleep*. Se qualquer uma das teclas for pressionada, uma interrupção ocorrerá, mesmo que seja uma única fonte. Nesse instante, o processador "acorda" e o programa continua sua execução, e não é possível saber qual das teclas que compartilham aquela entrada foi a responsável pela interrupção. O que pode ser feito então é um *polling*, checando todas as colunas exatamente como antes. Você pode imaginar que se o pressionamento for muito rápido, o tempo para que o sistema acorde e faça a verificação pode não ser suficiente para o programa detectar a tecla. No entanto, tudo isso ocorre em menos de 2 ms, e um acionamento mecânico humano é muito maior.

Figura 3.14 | Proposta de hardware para solução do problema



Fonte: elaborada pelo autor.

Figura 3.15 | Proposta de software para solução do problema

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/power.h>
#include <avr/sleep.h>
#define F_CPU 16000000UL
#include "util/delay.h"
#define TRUE 1

const unsigned char Disp7segLookup[12] = {
   0x3f, 0x06, 0x5b, 0x4f, 0x66,  //saídas
   0x6d, 0x7d, 0x07, 0x7f, 0x6f,  //acionadas em 1
   0x77, 0x78    // 'A' e 't'
};

void ConfigPCINT0(void){
    PCICR |= (1 << PCIE0);  //habilita interrupção  porta B
    PCMSK0 = 0x0F;  // hab int. canais 0 ao 3
}

unsigned char LeituraTeclado(void){
   unsigned char Tecla= 'n';   //variável local:
   Liga_C1();  _delay_ms(10);
   if(!(PINB & 0x01)) Tecla = '1';   if(!(PINB & 0x02)) Tecla = '4';
   if(!(PINB & 0x04)) Tecla = '7';   if(!(PINB & 0x08)) Tecla = '*';
   Liga_C2();  _delay_ms(10);
   if(!(PINB & 0x01)) Tecla = '2';   if(!(PINB & 0x02)) Tecla = '5';
   if(!(PINB & 0x04)) Tecla = '7';   if(!(PINB & 0x08)) Tecla = '0';
   Liga_C3();  _delay_ms(10);
   if(!(PINB & 0x01)) Tecla = '3';   if(!(PINB & 0x02)) Tecla = '6';
   if(!(PINB & 0x04)) Tecla = '9';   if(!(PINB & 0x08)) Tecla = '#';
   return Tecla;
}
void ConfigPORTS(void){
    PORTB = 0x3F;   //Saídas em nível 1 e
          //entradas com pull-up interno
    DDRB =  0x30;   //PB0 ao BP3 entradas (linhas)
          //e PB4 e PB5 saídas (colunas 1 e 2)
    PORTC = 0x01;  //desativa coluna 3 e led
    DDRC =  0x03;   //PC0 e PC1 saídas (col3 e led)
    PORTD = 0x00;   //desliga segmentos do display
    DDRD =  0x7F;   //PD0 ao BD6 saídas (disp 7-seg
}
#include <avr/cpufunc.h>
ISR (PCINT0_vect){   // rotina de tratamento PCINT0
   _NOP();    // (não faz nada, apenas acorda a CPU)
}
```

```
void Liga_C1(void){
   PORTC |= 0x01;
   PORTB |=0x30; PORTB &=0xEF;}

void Liga_C2(void){
   PORTB |=0x30; PORTB &=0xDF;}

void Liga_C3(void){
   PORTB |=0x30; PORTC &=0xFE;}

void Liga_CT(void){
   PORTB &=0xCF; PORTC &=0xFE;}

int main(void){      //função principal

   unsigned int CntVet[12] = {0,0,0,0,0,0,0,0,0,0,0,0};
   unsigned char Tecla;
   ConfigPORTS();
   ConfigPCINT0();
   sei();
   set_sleep_mode(SLEEP_MODE_PWR_DOWN);

   while(TRUE){   //Loop infinito
      PCIFR = 0xFF;  //limpa todas flags PCINT0
      Liga_CT();        //zera todas colunas..
      sleep_enable(); //para qqr tecla interromper
      sleep_cpu();   //  → Vai dormir
      //ZZZ...
      sleep_disable();      //acorda
      Tecla = LeituraTeclado();   //"varredura"
      if(Tecla!='n'){     //confirma tecla apertada
         PORTC |= 0x02;  //liga o led
         if(Tecla=='*')  Tecla=10; //  'A' sterisco
         else if(Tecla=='#')  Tecla=11;  //  't' ralha
         else Tecla -= 0x030; //conv char para num
         CntVet[Tecla]++;
          PORTD = Disp7segLookup[Tecla];
         _delay_ms(1000);
          PORTD = Disp7segLookup[CntVet[Tecla]];
         _delay_ms(1000);
         PORTC &= 0b11111101; //desliga o led
         PORTD = 0x00;   //desliga o display
      }
   }
}
```

Fonte: elaborada pelo autor.

Avançando na prática

Sistema de semáforo veicular com interrupção de pedestre

Descrição da situação-problema

Você é o técnico responsável por desenvolver um programa para sistema embarcado responsável por controlar um semáforo de veículos, através do ATmega328. Como especificações do seu novo projeto, tem-se: o conjunto possui três saídas, para as lanternas vermelha, amarela e verde, e uma entrada, que é o botão do pedestre, que tem prioridade sempre. Dessa forma, as luzes devem revezar conforme a sequência universal, com 25 segundos para a luz verde, 5 para a amarela e 30 para a vermelha. Caso o pedestre acione o botão, apenas considerado durante o sinal verde, o semáforo deve imediatamente ir para o sinal amarelo, "saltando" uma parte do ciclo, que continua depois normalmente. Seu chefe também instruiu a usar uma interrupção para atender o sinal do pedestre, para que a resposta seja imediata, e associar as entradas e as saídas a qualquer um dos canais digitais. Mãos à obra!

Resolução da situação-problema

Para a resolução, vamos considerar que as saídas para as lanternas verde, amarela e vermelha serão os canais PC0, PC1 e PC2. Para a entrada, vamos utilizar o canal PD2, correspondente à interrupção externa INT0. Como algoritmo, é possível que o programa se mantenha checando um contador para alterar o estado da máquina de estados, correspondente ao sinal acionado. Dessa forma, a interrupção gerada pelo acionamento do pedestre pode imediatamente acionar a lanterna amarela, e reajustar a variável de estado e o contador para o início do sinal amarelo. Quando o processamento retornar para a função principal, as variáveis alteradas serão lidas normalmente na próxima iteração, e o ciclo segue seu percurso a partir desse ponto.

Figura 3.16 | Programa proposto como solução para o problema

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define F_CPU 16000000UL
#include "util/delay.h"

#define TRUE 1
#define VRD  0
#define AMR 1
#define VRM 2

unsigned char VarESTADO = VRD, Cnt=0;
            //vars globais
void ConfigINT0(void){
    DDRD = 0x00;          // PD2 entrada
    PORTD |= (1 << PORTD2); //hab pull-up  PD2
    EICRA = 0x01; //INT0 acionada borda subida
    EIMSK |= (1 << INT0); // Habilita interrupção
}

ISR(INT0_vect){ //tratamento da interrupção INT0
    Cnt=0;
    VarESTADO = AMR;
    PORTC =  (1 << AMR);   //Liga lanterna amarela
}
```

```
int main(void){
    PORTC = (1 << VRD); //Liga lanterna verde
    DDRC = 0x07;
    //PC0, PC1 e PC2 saídas: VRD, AMR e VRM
    ConfigINT0();

    while(TRUE){
        switch(VarESTADO){
        case VRD:
            if(Cnt>24){
                Cnt=0;
                VarESTADO = AMR;
                PORTC =  (1 << AMR);   //Liga AMR
            } break;
        case AMR:
            if(Cnt>4){
                Cnt=0;
                VarESTADO = VRM;
                PORTC =  (1 << VRM);   //Liga VRM
            } break;
        default:
            if(Cnt>29){
                Cnt=0;
                VarESTADO = VRD;
                PORTC =  (1 << VRD);   //Liga VRD
            }
        }
        Cnt++;
        _delay_ms(1000);
    }
}
```

Fonte: elaborada pelo autor.

Faça valer a pena

1. Os microcontroladores mais modernos possuem mecanismos de operação que permitem o desligamento de apenas algumas partes do sistema, inclusive o núcleo (CPU), nos momentos que não há demanda de atividade. Isso permite a construção de soluções embarcadas que consomem pouquíssima energia, permitindo uma vida útil para as baterias de até anos.

Considerando as formas de se detectar o momento em que as tratativas devem ser feitas nos sistemas embarcados, observe as seguintes afirmações:

I – A técnica de *polling* utiliza exclusivamente funções para ser implementada, enquanto que a técnica de interrupção utiliza somente as rotinas de tratamento de interrupções.

II – A escolha sobre qual das técnicas deve ser usada para o tratamento de cada uma das tarefas não é feita livremente pelo programador, pois algumas tarefas exigem o uso de interrupções devido à sua própria natureza. Por

outro lado, algumas tarefas específicas se mostram impossíveis de serem tratadas por interrupção, exigindo o emprego do *polling*.
III – Se escritas igualmente, as duas técnicas são equivalentes, e geram o mesmo resultado de processamento. No entanto, a interrupção oferece algumas vantagens sobre o *polling*, que é uma menor latência média para resposta a uma requisição, e a possibilidade de poupar energia em modos de hibernação.
IV – Pode-se afirmar que na maioria dos casos, o *polling* é usado para tratar saídas e a interrupção para tratar entradas, pois estas foram criadas com esta finalidade.

Qual das alternativas a seguir apresenta corretamente a sequência de verdadeiro ou falso?
a) V, F, V, V.
b) V, V, F, F.
c) F, F, F, V.
d) F, V, V, V.
e) F, F, V, F.

2. Os aparelhos eletrônicos que funcionam sem conectividade direta à rede elétrica, ou seja, alimentados por bateria, devem ser construídos com muito cuidado para não consumirem energia excessiva. Para isso, é essencial que o núcleo do sistema pare de funcionar nos momentos de inatividade.
Considerando o sistema de interrupções presente nos microcontroladores, qual alternativa não pode ser considerada verdadeira?
a) As interrupções podem ser comparadas com as funções, pois são um conjunto de comandos agrupados em sequência. No entanto, diferente das funções, as rotinas de tratamento das interrupções não possuem parâmetros de entrada nem de saída, pois não são invocadas por nenhuma função.
b) As interrupções no ATmega328, como na maioria dos sistemas computacionais, são configuradas e habilitadas individualmente, e podem ser desativadas todas de uma vez, por uma "chave geral". Isso é feito no Atmega328 pelo bit GIE - *General Interrupt Enable*, que fica no registrador de status, no núcleo, e pode ser acionado em C por sei(); (*Set Globa Interrupt Flag*).
c) Todos os canais digitais do ATmega328 possuem fontes de interrupção exclusiva, permitindo que qualquer programa seja feito sem qualquer uso de *pooling*.
d) O Atmega328 possui seis modos de hibernação diferentes, além do modo ativo, e cada um corresponde a um conjunto de periféricos de

que deve se manter em atividade. Dessa forma, o programador tem mais flexibilidade de atender às necessidades mínimas, economizando energia sem perder desempenho.
e) Apesar de não ter o uso indicado para muitas quantidades, o sistema de interrupções do ATmega328 permite que mais de uma interrupção seja tratada concorrentemente. Não podem ser executadas em paralelo porque existe apenas um núcleo, mas podem ser interrompidas e permanecer aguardando em uma fila de prioridade para terminar.

3. O sincronismo de tarefas em sistemas embarcados começa a se mostrar um problema à medida que a complexidade aumenta, principalmente quando as exigências de latência na resposta são críticas. As interrupções mostram uma excelente forma de prover esse tipo de sincronismo.
Considerando o vetor de interrupções do microcontrolador ATmega328, bem como o seu sistema de interrupções, leia as seguintes afirmações:

I – O vetor de interrupções padrão do ATmega328 está mapeado nos primeiros endereços de memória de programa, em que cada par de endereços corresponde ao desvio para uma rotina de tratamento específica.
II – Existem duas formas de causar interrupções através dos canais digitais: as INTx e PCINTx. Na primeira, também chamada de interrupção externa, alguns poucos pinos possuem fontes exclusivas de interrupção, e até quatro formas de configuração. Na segunda, todos os canais de uma mesma porta compartilham uma única fonte de interrupção, e não há formas de se configurar o acionamento, que é causado por qualquer transição.
III – Esse modelo pertence a uma família de microcontroladores de baixo consumo de energia, pois permite a operação em modos de hibernação. No entanto, ele apresenta uma limitação, pois os canais de uma mesma porta não podem ser habilitados individualmente para gerar interrupção na fonte que é compartilhada. Dessa forma, se um *led* e um botão estão ligados a uma mesma porta, e se deseja que apenas o botão cause interrupção, não é possível fazer com que uma alteração no *Led* não cause uma também.
IV – Apesar da maioria das interrupções serem capazes de despertar o sistema, a partir de diferentes modos de hibernação, não é permitido que o microcontrolador passe por "longos períodos" sem ser despertado. Além disso, o programador deve se atentar para não permitir que mais de uma interrupção seja acionada, o que pode causar falhas na execução e comportamentos imprevisíveis. Isso pode ser feito desabilitando as interrupções globalmente no início da rotina de tratamento de todas as

interrupções (habilitando novamente no final).

Qual das alternativas a seguir apresenta corretamente a sequência de verdadeiro ou falso?

a) V, F, F, F.
b) F, F, F, V.
c) V, V, F, F.
d) V, F, V, F.
e) V, V, V, V.

Seção 3.3

Temporizadores

Diálogo aberto

Esse é sem dúvida um passo muito importante na nossa caminhada de aprendizado sobre os sistemas embarcados. Em algumas passagens, os temporizadores já foram mencionados, e brevemente comparados com as técnicas de sincronismo apresentadas, que utilizam rotinas de atraso. Agora você verá uma nova maneira de criar amarrações temporais entre eventos, de forma mais prática e precisa. Os temporizadores podem ser usados também para outras finalidades, mas seu papel básico é contabilizar um intervalo de tempo pré-definido, e avisar o programa principal quando aquele intervalo expirar. Isso permite que o processador exerça outras atividades enquanto o tempo passa. Assim como as interrupções externas, um dos módulos temporizadores também é capaz de despertar o processador da hibernação. Esse mecanismo é muito importante para a criação de sistemas de baixo consumo, que podem se manter inativos quando não precisam atuar, mas sem comprometer a latência de tratamento das ações. As interrupções servem para "despertar" o núcleo, ou outras partes do sistema que estejam "hibernando", ou seja, temporariamente inativas. Depois de entender como funciona o vetor de interrupções interno, e como as interrupções externas devem ser configuradas para determinadas aplicações, você deve resolver uma questão com esse assunto para entender melhor como deve atuar em novas soluções. Nessa situação, você trabalha em equipe para desenvolver um dispositivo de interface com o usuário de uma casa automatizada. Cada um da equipe é responsável por uma parte do projeto de automação doméstica, e a sua tarefa foi desenvolver a interface, tanto em hardware quanto em software, para o teclado matricial, que identificará para o sistema os comandos numéricos. Depois de ter desenvolvido os primeiros projetos, de maneira mais básica, nesta última situação aprimoraremos as rotinas de processamento, com o uso do módulo temporizador TIMER0, para prover um sincronismo

temporal mais avançado para o conjunto. Este deve atuar com as mesmas especificações da situação-problema anterior, com a diferença de não usar nenhuma função de atraso ("delay_ms(X)") para controlar o tempo entre ações, apenas com o temporizador e sua interrupção. Boa sorte!

Não pode faltar

Depois de ter estudado sincronismo de tarefas por meio de rotinas de atraso e manipulação de eventos por *polling* e interrupção, vamos estudar agora o periférico mais importante para prover sincronismo em sistemas computacionais, pois foi feito para isto. Veremos que os temporizadores, quando operando em paralelo, podem prover sincronismo avançado, mesmo para sistemas de tempo real crítico e com muitas tarefas realizadas em paralelo.

Assimile

O modelo Atmega328 compõe três módulos temporizadores: o TC0 - *Timer Counter 0*, o TC1 e o TC2.

Os temporizadores funcionam de maneira semelhante às variáveis/contadores, pois geram referência de tempo entre eventos a partir do valor presente em uma variável contadora. Por exemplo, se em um momento essa variável é zerada pelo programa, que em seguida a incrementa a cada 10 ms, é possível saber que se passaram 150 ms quando a variável atinge o valor 15. A diferença é que o temporizador oferece maiores benefícios, como melhor precisão, resolução, paralelismo e a capacidade de acordar o CPU em hibernação no momento certo de trabalhar. Para isso, os módulos temporizadores utilizam um registrador que armazena a contagem atual (análogo à variável/contador) que é incrementado até seu valor máximo, de acordo com uma frequência pré-configurada. Também há um outro registrador usado para o limite máximo de contagem (análogo à constante usada para comparação), ou seja, de maneira básica e genérica, um módulo temporizador pode ser usado para prover um controle temporal entre eventos a partir da configuração da frequência de contagem, o ajuste do máximo a

ser contado e a inicialização da contagem propriamente dita. Por exemplo, sabendo que o tratamento B deve ser executado 150 ms depois do tratamento A, é possível que o programa "agende" a rotina B através de um contador: configura a frequência de incremento para 1 KHz, o comparador (contagem máxima) para 150 e inicia a contagem. Como visto anteriormente, existem duas maneiras do programa usuário "saber" que a contagem terminou: por *polling* ou por interrupção. Você pode estar se perguntando: por que não adicionar simplesmente uma rotina de atraso de 150 ms entre as rotinas A e B? Apesar de funcionar, a ideia agora é deixar o CPU livre para trabalhar em outras tarefas, ao invés de se manter "congelado", aguardando o tempo passar. E, além disso, as tarefas que estão agendadas para serem tratadas não podem ser esquecidas, e devem ocorrer no instante correto. Isso é muito útil para utilizar o serviço do CPU apenas nos momentos oportunos, evitando desperdício de energia. Podemos comparar o temporizador com o relógio do nosso forno, ou micro-ondas. Ao invés de ficarmos na frente contando o tempo, podemos agendar o forno para apitar daqui a 20 minutos, e adiantar o trabalho de outras tarefas aguardando para serem tratadas.

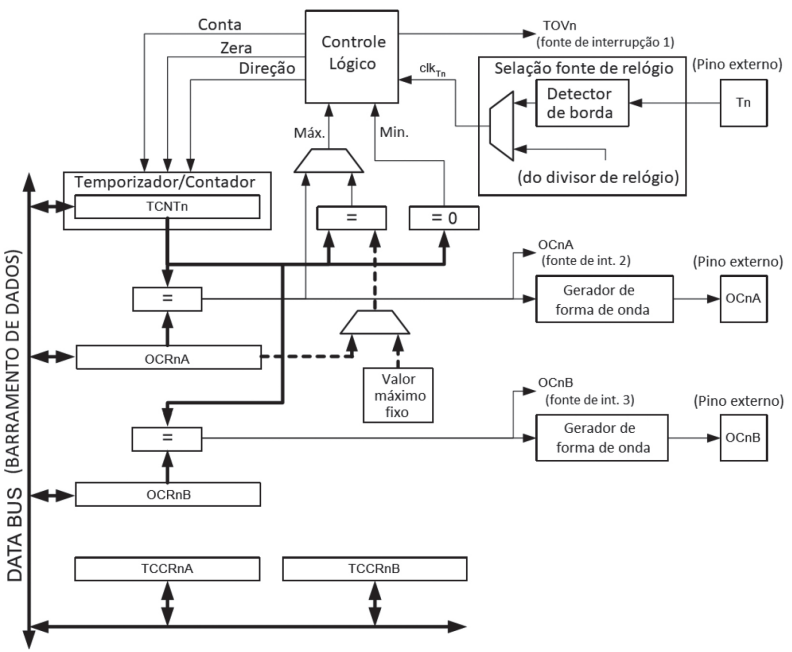
Todos os três módulos possuem a capacidade de gerar sinais PWM - *Pulse Witdth Modulation* (modulação por largura de pulso), que é uma maneira de controlar saídas de maneira analógica, ou com graduações, a partir de uma única saída digital. É possível controlar a velocidade de um motor, ou a intensidade de brilho de um *led*, por exemplo. Esse sinal se constitui de uma onda quadra (binível) de frequência constante, mas com ciclos positivos e negativos variáveis (complementares), como será visto mais adiante.

TC0 - Módulo Temporizador/Contador Zero

Esse módulo possui internamente duas unidades de comparação, com saídas parcialmente independentes, identificadas pelas letras A e B. Possui também dois registradores independentes de *Output Compare*, ou seja, comparadores para/de saída. Os módulos TC0A e TC0B não são totalmente independentes porque compartilham o mesmo registrador de contagem. Estes módulos, assim como seus registradores, são de 8 bit, e não trabalham com valores negativos, ou seja, a contagem pode ser feita de 0 até 255. Se for necessária a

contabilização de valores maiores, deve-se reduzir a frequência de contagem para que o intervalo temporal de interesse corresponda a um número menor que 255. Uma outra forma, até mais prática, é utilizar uma variável contadora, para detectar intervalos de tempo múltiplos do intervalo entre interrupções. Por exemplo: uma tarefa é executada depois de 5 interrupções do temporizador consecutivas. Uma grande vantagem deste módulo, em relação aos demais, é que possui um gerador de onda de relógio interno e próprio, que pode ser usado como fonte de contagem, assim como as fontes de relógio geral. Isso permite que os sinais gerais sejam desligados e o temporizador continue em atividade, nos momentos de hibernação. Vale ressaltar que apenas o módulo temporizador 2 possui a capacidade de despertar o processador em modo de hibernação, apesar de todos possuírem fontes de interrupção exclusivas. Podemos ver o seu diagrama interno na Figura 3.17:

Figura 3.17 | Diagrama interno do módulo temporizador zero do ATmega328

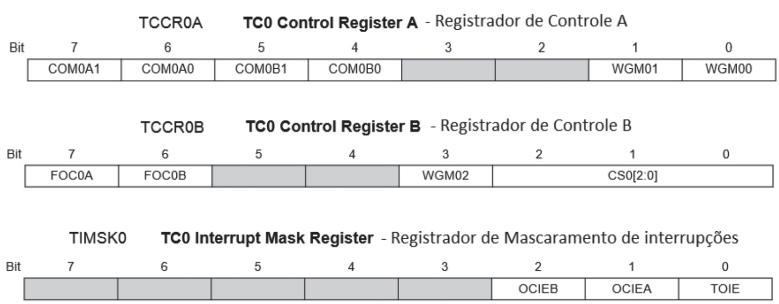


Fonte: elaborada pelo autor.

Registradores do Temporizador Zero

O registrador responsável pela contagem é o **TCNT0** - *Timer/Counter 0 Register*, que é incrementado a cada pulso de contagem, e comparado com os registradores TC0A e TC0B, para controlar as duas saídas. Este módulo pode gerar interrupções através de três fontes distintas: **OCF0A**, **OCF0B** e **TOV0**, as quais podem ser mascaradas individualmente através dos bits do registrador **TIMSK0** - *Timer Interrupt Mask Register Zero*. Estas interrupções, quando ocorrem, acionam os respectivos bits/*flags* no registrador **TIFR0** - *Timer Interrupt Flag Register*, que geram as interrupções que estão habilitadas (não mascaradas). Estas *Flags* também são utilizadas pelo programa usuário para saber se o temporizador terminou a contagem de tempo programada, através do *polling*. O temporizador zero pode ser "alimentado" por um sinal de relógio (*clock*) interno através de um submódulo chamado preescaler (divisor de relógio), ou mesmo por um sinal de relógio externo ao chip, através do pino T0. Os registradores são vistos na Figura 3.18, e suas configurações, na figura seguinte:

Figura 3.18 | Registradores de configuração do temporizador zero



Fonte: elaborada pelo autor.

Figura 3.19 | Configuração dos registradores do temporizador zero

Waveform Generation Mode Bit Description - Descrição dos bits para geração de forma de onda

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCR0x at	TOV Flag Set on[1),(2)]
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	-	-	-
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	-	-	-
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Note: 1. MAX = 0xFF 2. BOTTOM = 0x00

Clock Select Bit Description - Descrição dos bits para seleção do sinal de relógio

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	clkI/O/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Compare Output Mode, non-PWM - Modos de comparação de saída, para não PWM

COM0A1	COM0A0	Description
0	0	Normal port operation, OC0A disconnected.
0	1	Toggle OC0A on Compare Match.
1	0	Clear OC0A on Compare Match.
1	1	Set OC0A on Compare Match .

Fonte: elaborada pelo autor.

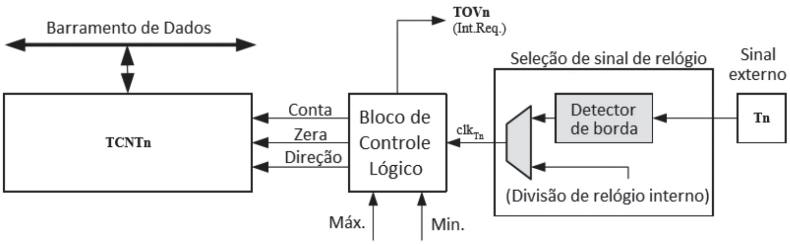
Fontes de relógio

A fonte de relógio para a contagem é selecionada através do bloco de controle lógico de seleção de fonte de relógio, que também decide se as bordas do sinal serão usadas para incrementar, ou decrementar o valor de contagem. Para desligarmos o Temporizador Zero, basta não selecionar alguma das fontes de relógio. A fonte de relógio TC0 é selecionada através dos três primeiros bits do registrador **TCCR0B** - *Timer/Counter Control Register*. Note que a letra B aqui não se refere à parte B do módulo interno, mas ao segundo registrador de controle do periférico.

Unidade de contagem

Uma característica importante do registrador de contagem, o TCNT0, é que ele é autogerenciável, mas também pode ser escrito pelo programa usuário. Desta forma, caso haja necessidade, o programa usuário pode modificar o valor atual de contagem, além de ler. O bloco da unidade de controle pode ser visto na Figura 3.20:

Figura 3.20 | Unidade de contagem do temporizador zero



Fonte: elaborada pelo autor.

A cada pulso de sinal de *clock*, o contador pode ser zerado, incrementado ou decrementado, dependendo de qual modo de operação é usado. A sequência de contagem é determinada pela configuração dos bits WGM01 e WGM00, localizados no registrador TCCR0A, além do bit WGM02, no registrador de controle B. A *flag* de interrupção TOV0 - *Timer/Counter Overflow Flag* é acionada de acordo com o modo de operação selecionado através dos três bits WGM0x recém-citados, e pode ser usado para gerar interrupções no processamento.

Reflita

Se for utilizado o sinal de relógio interno, de 16MHz, para a contagem do temporizador zero do ATmega328, é possível obter interrupções com intervalos de 100 ms, mesmo com os divisores de relógio? Se não, como é possível tratar uma tarefa a cada 100 ms (ou mais) com este periférico interno, nessas condições?

Unidade de Comparação de Saída

Por estar implementada em hardware, essa unidade executa a comparação continuamente entre o valor corrente de contagem, em TCNT0, e os registradores de comparação OCR0A e OCR0B. Assim que os valores se igualam, um sinal de correspondência (*match*) é acionado através das *flags* OCF0A e OCF0B, um ciclo de relógio depois. Este mesmo sinal é responsável por acionar uma interrupção, caso esteja habilitada. Essas *flags* são automaticamente zeradas depois que a interrupção é gerada, mas também pode ser zerado pelo programa usuário, escrevendo "1" no endereço correspondente.

O gerador de forma de onda utiliza o sinal de correspondência para gerar a onda de saída de acordo com o modo de operação (configurado pelos bits WGM0x), além do modo de comparação de saída, ditado pelos bits COM0A[1:0] e COM0A[1:0]. Os sinais de máximo e mínimo são usados pelo gerador de forma de onda para casos de valores extremos.

Modos de operação

Estes modos determinam o comportamento do temporizador zero, bem como os pinos correspondentes ao *Output Compare*, através da combinação do modo de geração de forma de onda, e o modo de comparação de saída.

Modo Normal - é considerado o modo mais básico, em que a contagem é sempre crescente, e não ocorre o zeramento automático do contador antes deste atingir o seu limite (255). Assim que o contador "estoura", ou seja, está com 255, é incrementado e retoma o valor zero, a *flag* de estouro TOV1 é acionada. Note que o registrador de comparação não é utilizado neste modo, o que torna a frequência de incremento o único responsável pelo intervalo de tempo entre interrupções sucessivas.

Modo CTC - *Clear Timer on Compare Match* (zera o temporizador quando a comparação é equivalente). Este modo permite um ajuste mais flexível para o intervalo de tempo desejado, pois a contagem pode ser feita de zero até qualquer valor menor do que 256, além da escolha da frequência de contabilização. Se a saída da comparação estiver habilitada, esta será invertida cada vez que o contador for reiniciado.

Modo PWM rápido - é configurado através de: WGM0 [2: 0] = 0x3 ou = 0x7) fornece uma opção de geração de forma de onda PWM de alta frequência. Os modos *Fast PWM* diferem das outras opções PWM por sua operação de inclinação única, ou seja, o sentido da contagem não é invertido quando esta atinge o seu valor máximo. O contador conta de zero até máximo, depois reinicia em zero. O valor máximo é definido como 0xFF com WGM0 [2: 0] = 0x3, e como OCR0A para WGM0 [2: 0] = 0x7. No modo de *Output Compare* não invertido.

Configuração do módulo temporizador zero

Podemos dividir este procedimento nos seguintes passos:

1- A primeira ação é escolher o modo de operação do temporizador, através dos bits WGM0x. Note que, para o modo normal (contando de 0 a 255, sem comparação e PWM), não é necessário fazer nada, pois já é o modo padrão. Para o modo CTC, que provê um melhor ajuste do tempo entre interrupções, o segundo bit deve ser acionado, como foi mostrado anteriormente.

2 - Em seguida, deve-se escolher a fonte de relógio para a contagem, se será interna, com ou sem divisor, ou externa, por um sinal de relógio de entrada no pino T0. Esta é a ação que habilita e inicia o temporizador, pois seleciona e conecta a fonte de relógio, através dos bits CS0x. O seu valor padrão é 000, ou seja, nenhuma fonte de relógio.

3 - Caso o modo escolhido seja o CTC ou PWM, os registradores OCR0A e OCR0B devem ser devidamente carregados.

4 - Se for definido que a aplicação utilizará alguma interrupção do temporizador zero, estas devem ser habilitadas pelos bits de mascaramento, além de terem suas rotinas descritas e identificadas no programa principal.

Exemplificando

Vamos escrever um programa para que o *led*, presente na placa Arduino UNO, pisque em 1 Hz (aproximadamente) através do temporizador zero, no modo normal, e através de interrupção. Para isso, devemos fazer algumas regras de três para encontrar os valores de configuração. Se utilizarmos a fonte de relógio interna, que é de 16 MHz, com um divisor de 1024, uma interrupção será gerada a cada (1024/16MHz)*256 = 16,384 ms. Esse não é o intervalo desejado, e nem um divisor comum, mas se utilizarmos uma variável para contabilizar 30 interrupções seguidas, é possível inverter o estado do led a cada 0,49152 segundos. Como visto na Figura 3.21:

Figura 3.21 | Exemplo para o ATmega328 com o temporizador 0

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define TRUE 1

volatile char FlagTMR0;
unsigned int CntTMR0;
   //variáveis globais

ISR(TIMER0_OVF_vect) {
   if(CntTMR0>=30){
      CntTMR0 = 0;
      FlagTMR0 = 1;
   }else
      CntTMR0++;
}
```

```
int main(void) {

   DDRB = 0b00100000;
      //PB5 saída -> Led

   TCCR0B = (1 << CS02) | (1 << CS00);
      //Seleciona fonte interna e com
      //divisor de relógio de 1024
   TIMSK0 |= (1 << TOIE0);
      //Habilita interrupção
      //para o modo normal

   sei(); //habilita interrupções globais

   while(TRUE){

      if(FlagTMR0){
         PORTB ^= (1 << PB5);
            //inverte o estado do led
         FlagTMR0 = 0;
      }
   }
}
```

Fonte: elaborada pelo autor.

Podemos observar que, no exemplo anterior, a contagem de tempo não é precisa, e não há tarefas paralelas, ou seja, uma rotina de *delay* faria o mesmo trabalho. Para melhor controle dos intervalos de tempo, é importante o uso do modo CTC.

Exemplificando

Neste novo exemplo, é requerido uma frequência de piscadas do led em exatamente 1 Hz. Se utilizarmos a fonte de relógio interna (para o temporizador zero) de 16 MHz, com um divisor de 64, e o valor máximo de contagem 249, uma interrupção será gerada a cada (64/16MHz)*250 = 1 ms. A partir deste valor, é possível definir qualquer intervalo preciso em milissegundos, através de uma variável/contador, visto na Figura 3.22:

Figura 3.22 | Exemplo para o ATmega328 com o temporizador 0

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define TRUE 1

volatile char FlagTMR0;
unsigned int CntTMR0;
   //variáveis globais

ISR(TIMER0_COMPA_vect){
   CntTMR0++;
   if(CntTMR0>=500){
      CntTMR0 = 0;
      FlagTMR0 = 1;
   }
}
```

```
int main(void) {

   DDRB = 0b00100000;
      //PB5 saída -> Led

   TCCR0A = (1 << WGM01);
      //Seleciona modo CTC
   OCR0A = 249;
      //valor máximo de contagem
   TCCR0B = (1 << CS01) | (1 << CS00);
      //Seleciona fonte interna e com
      //divisor de relógio de 64
   TIMSK0 |= (1 << OCIE0A);
      //Habilita interrupção
      //para o modo CTC
   TCNT0 = 0; //zera o contador
   sei(); //habilita interrupções globais

   while(TRUE){

      if(FlagTMR0){
         PORTB ^= (1 << PB5);
            //inverte o estado do led
         FlagTMR0 = 0;
      }
   }
}
```

Fonte: elaborada pelo autor.

Podemos observar que, mesmo com o maior divisor de relógio (1024), é impossível detectar períodos mais longos sem a utilização de variáveis/contadores.

Pesquise mais

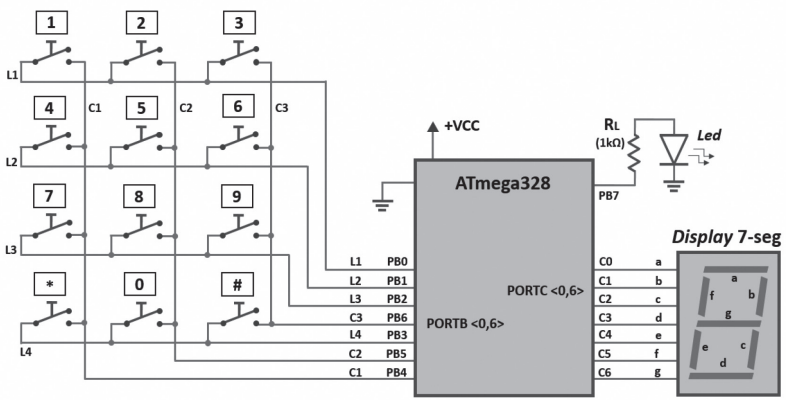
Os demais módulos temporizadores, além de poderem operar paralelamente, possuem algumas características particulares, como o uso de 16-bit para o temporizador 1, e a capacidade de interromper e retirar o processador da hibernação, para o 2. No link a seguir, essas informações estão muito bem explicadas e exemplificadas. Disponível em: <https://www.embarcados.com.br/timers-do-atmega328-no-arduino>. Acesso em: 27 set. 2017.

Sem medo de errar

Este último desafio da unidade, apesar de não apresentar nenhuma necessidade crítica para precisão de tempo, serve como um ótimo exemplo para essa finalidade. Vamos agora resolver o problema proposto, começando com uma análise sobre suas especificações técnicas. Antes de tudo, vamos relembrar o que deve ser feito. Nessa situação, você trabalha em equipe para desenvolver um dispositivo de interface com o usuário de uma casa automatizada. Cada um da equipe é responsável por uma parte do projeto de automação doméstica, e a sua tarefa foi desenvolver a interface, tanto em hardware quanto em software, para o teclado matricial, que identificará para o sistema os comandos numéricos. Portanto, você fará agora um sistema "teste", que é autossuficiente, para que depois possa ser incluído no protótipo inicial. Sabendo que utilizaremos um teclado numérico matricial para esse projeto, algumas decisões devem ser tomadas para realizar a conexão entre os sinais das teclas e o microcontrolador Atmega328. Nesse novo cenário, o display deve apresentar dois valores para cada acionamento de botão, com um segundo de duração cada. O primeiro valor representa a tecla pressionada, como no caso anterior, e o Led1 deve permanecer aceso juntamente. Logo após, deve ser exibido a quantidade de vezes que aquela tecla foi apertada desde a última inicialização do conjunto, e apenas o Led2 deve acender, e depois do intervalo, tudo se apaga. Acontece que agora o sistema deve detectar um acionamento enquanto exibe um outro anterior, dessa forma, se no início a tecla 5, por exemplo, for pressionada duas vezes rapidamente, o segundo valor que deve aparecer é o 2, omitindo a exibição para o primeiro acionamento. Por usar interrupção, o sistema pode não fazer nada no loop principal (apenas nas rotinas de interrupção), ou realizar as devidas tratativas através do uso "*flags*", acionadas na interrupção. Nessa última situação, aprimoraremos as rotinas de processamento, com o uso do módulo temporizador TIMER0, para prover um sincronismo temporal mais avançado para o conjunto. Este deve atuar com as mesmas especificações da situação-problema anterior, com a diferença de não usar nenhuma função de atraso ("delay_ms(X)") para controlar o tempo entre ações, apenas com o temporizador e sua interrupção. Para resolver as particularidades desta tarefa, podemos usar o temporizador, calibrado para interromper a cada 1 ms, para

simplesmente incrementar uma variável global. Esse pode ser um dos métodos mais simples sem se utilizar o temporizador. Isso pode ser visto nas Figuras 3.23, para o hardware, e 3.24 para o software.

Figura 3.23 | Circuito como solução para a situação-problema



Fonte: elaborada pelo autor.

Figura 3.24 | Programa como solução para a situação-problema

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define TRUE 1
#define LIGA_C1 {PORTB |=0x70; PORTB &=0xEF;}
#define LIGA_C2 {PORTB |=0x70; PORTB &=0xDF;}
#define LIGA_C3 {PORTB |=0x70; PORTB &=0xBF;}
#define LIGA_CT {PORTB &=0x8F;}

const unsigned char Disp7segLookup[12] = {
   0x3f, 0x06, 0x5b, 0x4f, 0x66,       //saídas acionadas em 1
   0x6d, 0x7d, 0x07, 0x7f, 0x6f,
   0x77, 0x78   // 'A' e 't'
};

unsigned int CntTMR0;

ISR(TIMER0_COMPA_vect){
    CntTMR0++;
}

unsigned char LeituraTeclado(void){
   unsigned char Tecla= 'n';   //variável local:
                               // n de "nenhuma"
  CntTMR0 = 0;
  LIGA_C1;  while(CntTMR0<10);  CntTMR0 = 0;
  if(!(PINB & 0x01)) Tecla = '1';   if(!(PINB & 0x02)) Tecla = '4';
  if(!(PINB & 0x04)) Tecla = '7';   if(!(PINB & 0x08)) Tecla = '*';
  LIGA_C2;  while(CntTMR0<10); CntTMR0 = 0;
  if(!(PINB & 0x01)) Tecla = '2';   if(!(PINB & 0x02)) Tecla = '5';
  if(!(PINB & 0x04)) Tecla = '7';   if(!(PINB & 0x08)) Tecla = '0';
  LIGA_C3;  while(CntTMR0<10);
  if(!(PINB & 0x01)) Tecla = '3';   if(!(PINB & 0x02)) Tecla = '6';
  if(!(PINB & 0x04)) Tecla = '9';   if(!(PINB & 0x08)) Tecla = '#';
  return Tecla;
}
```

```
int main(void){     //função principal

  unsigned int CntVet[10] = {0,0,0,0,0,0,0,0,0,0};
  unsigned char Tecla;

  PORTB = 0x7F;    //Saídas em nível 1 e entradas pull-up
                     //Led em PB7 desligado
  DDRB =  0xF0;    //PB0 ao BP3 entradas e demais saídas
  DDRC =  0xFF;    //PortC saída:  Display 7 seg. C0→ a, C6→g
  TCCR0A = (1 << WGM01); //Seleciona modo CTC
  OCR0A = 249;     //valor máximo de contagem
  TCCR0B = (1 << CS01) | (1 << CS00);  //Selec. fonte interna e
                                       //com divisor de relógio de 64
  TIMSK0 |= (1 << OCIE0A);    //Habilita int para o modo CTC
  TCNT0 = 0; //zera o contador
  sei(); //habilita globalmente as interrupções

  while(TRUE){   //Loop infinito

     Tecla = LeituraTeclado();   //realiza leitura por "varredura"
     if(Tecla!='n'){     //checa se tecla foi apertada
        PORTB |= 0x80;   //liga o led
        if(Tecla=='*')  PORTC = Disp7segLookup[10];
        else if(Tecla=='#')  PORTC = Disp7segLookup[11];
        else{
           Tecla -= 0x030; //converte char para num
           CntVet[Tecla]++;
           PORTC = Disp7segLookup[Tecla];
           CntTMR0 = 0;  while(CntTMR0<1000);
           PORTC = Disp7segLookup[CntVet[Tecla]];
           CntTMR0 = 0;  while(CntTMR0<1000);
        }
     }
  }
}
```

Fonte: elaborada pelo autor.

Perceba que o método de *debounce* está implicitamente inserido no programa, uma vez que, quando o primeiro pulso da sequência é detectado, o programa entra numa fase de escrita do display, que dura 2 segundos. Dessa forma, os próximos pulsos que chegam em sequência são seguramente ignorados pelo sistema.

Avançando na prática

Tranca eletrônica temporizada

Descrição da situação-problema

Você e sua equipe de desenvolvimento de sistemas embarcados perceberam que o laboratório é trancado apenas por chave, e decidiram construir uma maneira simples de abrir a porta, mas que apenas aqueles que conhecem o mecanismo podem fazer. A ideia adotada foi utilizar apenas um único botão, que deve ser pressionado ininterruptamente durante um intervalo entre 3 e 4 segundos, para que a porta seja aberta. Enquanto um dos companheiros elaborará o circuito, outro construirá a placa, e você deve escrever o programa a ser embarcado no projeto. Sabendo que o botão está conectado no canal PB0, sem resistor de *pull-up/down* externo, e que a saída para acionar o relé que abre a porta está no pino PC0, você deve fazer o programa conforme o combinado, através do temporizador zero do ATmega328.

Resolução da situação-problema

Uma solução básica, considerando a utilização do temporizador zero para prover referência de tempo para o programa, é simplesmente incrementar uma variável global a cada interrupção, que pode ser configurada para ocorrer a cada exatamente 1 ms. Dessa forma, o programa usuário pode controlar essa variável através de habilitação, zeramento e testes com constantes. Isso pode ser visto no programa da Figura 3.25:

Figura 3.25 | Programa proposto como solução para o problema

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define TRUE 1

unsigned int CntTMR0;

ISR(TIMER0_COMPA_vect){  // ocorre a cada 1ms
   if(CntTMR0<5000) CntTMR0++;
}

int main(void){      //função principal

   PORTB = 0x01;    //PB0 entrada pull-up interno
   DDRC =  0x01;    //PC0 saída:  Rele para a porta

   TCCR0A = (1 << WGM01); //Seleciona modo CTC
   OCR0A = 249;     //valor máximo de contagem
   TCCR0B = (1 << CS01) | (1 << CS00);  //Selec. fonte interna e
                                        //com divisor de relógio de 64
   TIMSK0 |= (1 << OCIE0A);    //Habilita int para o modo CTC
   TCNT0 = 0; //zera o contador

   while(TRUE){   //Loop infinito
      while(PINB & 0x01);   //aguarda botão ser acionado
      CntTMR0 = 0; //zera contador
      sei(); //habilita globalmente as interrupções
      while(CntTMR0<200);  //aguarda periodo de debounce
      CntTMR0 = 0; //zera contador
      while(!(PINB & 0x01));      //aguarda botão ser solto
      if((CntTMR0 > 3000)&&(CntTMR0 <4000)){
         CntTMR0 = 0; //zera contador
         PORTC = 0x01; //aciona rele que abre a porta
         while(CntTMR0<50);  //aguarda periodo
         PORTC = 0x00; //desliga rele que abre a porta
      }
      cli();  //desabilita globalmente as interrupções
   }
}
```

Fonte: elaborada pelo autor.

Faça valer a pena

1. É comum entre os módulos temporizadores internos aos microcontroladores, a separação entre subunidades de temporização para gerar sinais de PWM (modulação por largura de pulso), por exemplo.
A respeito do módulo interno ao microcontrolador ATmega328 temporizador zero, considere as seguintes afirmativas:

I – Esse módulo pode ser considerado como duplo, pois possui duas unidades de temporização completamente independentes, com dois registradores de comparação, e dois registradores de contagem independentes.
II – Apesar de existir mais de uma fonte de interrupção associada a este módulo, não é possível que estas sejam habilitadas de maneira independente, apenas juntas.
III – Este temporizador pode ser diretamente acessado pelo programa usuário, através de seus registradores, e só pode ser usado para tratar tarefas através de interrupção, não permitindo a manipulação via *polling.*

IV – As duas partes deste módulo podem ser utilizadas em conjunto, compartilhando fontes de interrupção, e somando os seus contadores individuais para prover contagens maiores.
V – Este é o único dos módulos temporizadores que não possui a capacidade de gerar sinais de modulação por largura de pulso, ou PWM.

Qual das alternativas a seguir apresenta corretamente a sequência de verdadeiro ou falso?
a) V, F, V, V, F.
b) F, F, V, F, V.
c) V, F, F, F, V.
d) F, F, F, F, F.
e) F, V, V, F, F.

2. Os módulos temporizadores dos microcontroladores, assim como os demais módulos, possuem como interface de comunicação o programa usuário, ou o CPU, através dos seus registradores.

A respeito do modulo temporizador zero presente no ATmega328, quais são os bits responsáveis pela configuração do divisor de relógio interno?
a) WGM0X.
b) CS0x.
c) COM0Ax.
d) OCIEx.
e) FOC0x.

3. Assim como os demais microcontroladores, o ATmega328 possui mais de um módulo temporizador interno, e com diferentes funcionalidades. Considerando a unidade de contagem do temporizador zero, presente no microcontrolador ATmega328, qual das seguintes alternativas contém uma sentença verdadeira?
a) Esta unidade é a única parte não autônoma do módulo temporizador zero, e deve ser gerenciada continuamente pelo programa usuário.
b) Agrega todas as funcionalidades da unidade de comparação de saída, uma vez que a contém.
c) Não é autossuficiente, pois não possui unidade de controle própria, mas já contempla o circuito digital divisor de relógio em seu interior.
d) Possui um gerador de sinal de relógio interno, acionado por um oscilador RC (Resistor-capacitor).
e) É autônoma e não possui acesso direto ao programa usuário, através da CPU, apesar de compartilhar com este o registrador de contagem, o TCNT0, e pode ser desabilitada pela ausência de sinal de relógio.

Referências

ATMEL AVR. **8-bit AVR Microcontrollers. ATmega328/P. Datasheet Complete**. San Jose: 2016. Disponível em: <http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf>. Acesso em: 27 set. 2017.

MARGUSH, T. S. **Some assembly required:** assembly language programming with the AVR microcontrollers. 1. ed. New York: CRC Press, 2011.

MAZIDI, M. A.; NAIMI S.; NAIMI S. **The avr microcontroller and embedded system:** using assembly and c. 1. ed. New Jersey: Prentice Hall, 2011.

LIMA, C. B. D; VILLAÇA, M. V. M. **AVR e arduino técnicas de projeto.** Florianópolis: [s.n], 2012.

Periféricos complexos

Convite ao estudo

Entramos agora na fase final do nosso caminho de aprendizado no universo dos microcontroladores. Com os conhecimentos adquiridos nas unidades anteriores, a partir do estudo dos periféricos básicos de seis registradores, estamos prontos para estudar alguns dos periféricos avançados presentes nesses dispositivos. Compreenderemos como podemos expandir as aplicações de nosso microcontrolador, além das interações digitais básicas que vimos até agora.

Entendemos que o microcontrolador desempenha um papel central na eletrônica moderna, mas, sendo um dispositivo de essência digital, necessita de periféricos específicos para a captação e para o processamento de sinais analógicos, que possuem características diversas daqueles que vimos até aqui, através dos sinais digitais. O papel do conversor analógico-digital, ou ADC (termo em inglês para *Analog-Digital* Converter), é tratar esses sinais e convertê-los em informação digital, para que o núcleo de nosso microcontrolador possa entender e manipular esse dado de entrada.

Apesar de extremamente versátil como um dispositivo integrado, o microcontrolador, em muitos projetos, não é suficientemente capaz de processar todos os sinais de entrada e saída em um circuito eletrônico. Somado a isso, após a conclusão de um projeto eletrônico embarcado, dada a sua complexidade, é essencial que um canal de comunicação entre o microcontrolador e um sistema externo para diagnose ou coleta de dados seja implementado. Veremos como utilizar os periféricos avançados de comunicação síncrona e assíncrona do microcontrolador para realizar a troca de informação

necessária entre ele e outros dispositivos do projeto, e também como poderemos estabelecer um canal de comunicação entre o nosso projeto e outro meio externo.

Os circuitos eletrônicos modernos devem ser projetados de forma a operar em condições adversas, e uma das situações mais comuns entre essas intempéries é falta de alimentação, repentina ou programada. Como os microcontroladores tratam essencialmente de fluxo de informação, todos aqueles dados presentes na memória volátil serão perdidos nessas situações. Desta forma, é preciso mapear quais dados devem ser armazenados e quais devem ser resguardados. Assim, estudaremos a implementação de mais um tipo de memória: a EEPROM (*Electrically-Erasable Programmable Read-Only Memory*), que permite guardar informações por longos períodos de tempo, mesmo na ausência de alimentação do circuito.

Para concretizar esses conceitos, trabalharemos sobre um projeto em que você é o responsável técnico pelo desenvolvimento de um sistema **de controle de temperatura com funções avançadas de aquisição de dados**, interface com o usuário e com os técnicos de campo, e também possuirá a capacidade de armazenar valores de medição e condições gerais do sistema em uma memória não volátil, de forma que o sistema permaneça estável e robusto mesmo nas situações mais adversas.

Nesta reta decisiva de nosso aprendizado, teremos a oportunidade de expandir nossos conhecimentos para além da pura manipulação digital que os periféricos básicos nos proporcionam e, ao final, estaremos prontos para implementar sistemas embarcados completos e totalmente funcionais. Ótimos estudos!

Seção 4.1

Conversor analógico-digital

Diálogo aberto

Apesar de o mundo ser dominado pelas tecnologias chamadas digitais, a maioria das variáveis presentes nele é de natureza analógica. Dessa forma, é tarefa dos sistemas embutidos de microprocessadores ou microcontroladores a conversão dessas informações analógicas em informação digital, para que se adéquem à língua que o nosso dispositivo compreende: a linguagem binária. Nesta seção, entenderemos primeiramente como caracterizar um sinal analógico típico, como temperatura, tensão elétrica, pressão, entre outros, e convertê-lo ao universo digital através da configuração adequada do nosso microcontrolador.

Como responsável técnico de uma empresa que desenvolve sistemas avançados de controle industrial, você foi designado para liderar o desenvolvimento do novo produto da empresa: um sistema de controle de temperatura. Nesse projeto, foi requisitado que fossem utilizadas duas entradas analógicas: a) um sensor de temperatura MCP9700 (escolhido pelo cliente), que pode ser disposto próximo ao objeto ou imerso ao ambiente onde se deseja realizar o controle e b) um potenciômetro, que será a interface utilizada para que o usuário do sistema possa configurar a temperatura desejada para o controle entre 25 ºC e 100 ºC. De forma a possibilitar o resfriamento do sistema quando a temperatura lida for superior a um determinado valor configurado, foi também requisitado pelo cliente o controle de uma ventoinha. Desta maneira, sempre que o valor real de temperatura for superior à temperatura configurada, a ventoinha deve ser ligada, e desligará na situação oposta. Para indicar o valor atual de temperatura, no caderno de especificações do projeto, é solicitada a inclusão de um led simples, que piscará em uma frequência diretamente proporcional à temperatura medida pelo sensor, assim, o usuário do sistema conseguirá monitorar a temperatura em tempo real.

Parece desafiador, não é mesmo? Não se preocupe, garantimos que com as informações e as técnicas que aprendemos até aqui, somadas a esse novo entendimento sobre sistemas de conversão

analógico-digital, conseguiremos desenvolver mais esse problema e dominar um dos sistemas de controle mais utilizados na engenharia! Vamos lá!

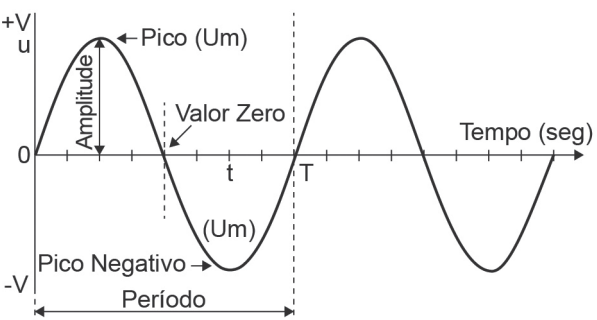
Não pode faltar

A natureza dos sinais analógicos e suas características

Até este momento, temos focado nosso estudo sobre como utilizar o microcontrolador para manipular entradas e saídas exclusivamente digitais, ou seja, aquelas que possuem somente dois níveis de tensão dentro de um intervalo de tempo, porém, grande parte dos sinais que um microcontrolador deve assimilar e controlar são de natureza analógica, o que significa que tais sinais podem assumir infinitos valores dentro de um intervalo de tempo. Entender o comportamento desses sinais e caracterizá-los é o primeiro passo para configurar de maneira adequada o microcontrolador que fará sua leitura.

Diversos são os dispositivos e os meios que interagem uns com os outros através de sinais analógicos. Dentre os mais comuns, pode-se destacar o sinal senoidal da tensão elétrica que alimenta os aparelhos da sua casa e os sensores de temperatura, que são muito utilizados em diversos equipamentos, seja para proteção dos circuitos, como sensor do ambiente, ou até mesmo para projetos de segurança (Figura 4.1).

Figura 4.1 | Sinal analógico senoidal



Fonte: <https://goo.gl/fHJw4A>. Acesso em: 23 out. 2017.

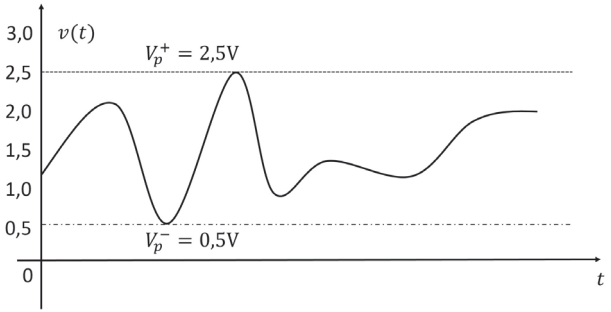
São inúmeros os atributos que podemos utilizar para caracterizar um sinal analógico, dentre os quais podemos destacar: grandeza do sinal (tensão, resistência elétrica, temperatura etc.), amplitude, frequência (quando cíclico) e histerese. No entanto, para a configuração do conversor analógico-digital, nos ateremos à natureza do sinal e a sua amplitude. Os conversores analógico-digital só podem converter valores de tensão elétrica, logo, caso o sinal seja de qualquer outra natureza, deve ser convertido para tensão antes de ser lido pelo conversor, como seria o caso do sensor de temperatura da Figura 4.1, em que sua grandeza é a resistência elétrica, variando de maneira inversamente proporcional à temperatura. A amplitude do sinal é outro atributo que requer atenção, pois os conversores operam em faixas preestabelecidas de tensão, e o sinal de leitura deve se encaixar dentro dessas faixas para que o valor possa ser lido de maneira satisfatória. Quando o sinal ultrapassa essas faixas, um circuito condicionador de sinal deve ser aplicado para normatizar o sinal.

Pesquise mais

Em muitos casos, precisamos adequar um sinal analógico de interesse para as características e para as limitações que o nosso circuito de leitura impõe. Geralmente, utilizamos associações entre componentes passivos (resistores, capacitores) e alguns amplificadores para construir um bom condicionador de sinais. Leia mais sobre esse tema em: Trazendo o mundo real para dentro do processador - Condicionamento de sinais analógicos: Disponível em: <https://www.embarcados.com.br/condicionamento-de-sinais-analogicos>. Acesso em: 23 out. 2017.

Veja como exemplo o sinal presente na Figura 4.2. Este é um sinal analógico genérico, cuja grandeza já é a tensão elétrica, ou seja, não necessitaria de um transdutor de sinal para a conversão de grandezas. Perceba também que ele varia livremente entre uma faixa de valores, e aos limites dessa faixa damos o nome de Valores de Pico, podendo ser de pico positivo (Vp+) ou de pico negativo (Vp-). Esses valores serão muito úteis para a configuração de nosso conversor.

Figura 4.2 | Sinal analógico genérico

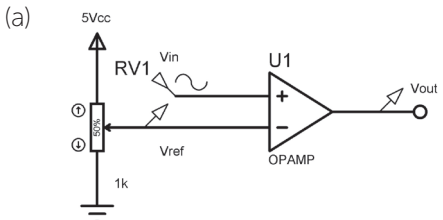


Fonte: elaborada pela autora.

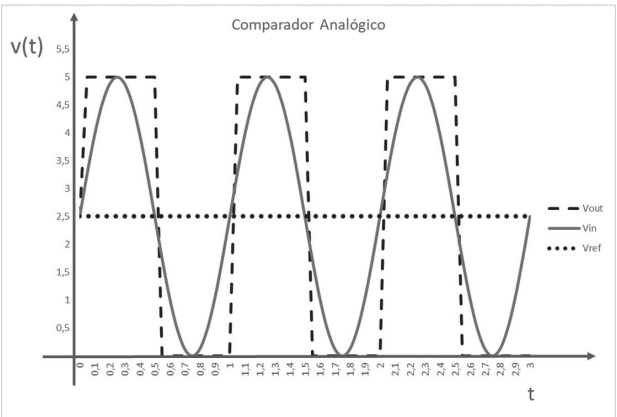
Circuitos conversores de sinais analógicos

Um circuito conversor analógico para digital opera basicamente através da aplicação de comparadores de tensão. Esses circuitos são implementados utilizando, por exemplo, amplificadores operacionais, em que o sinal de leitura é conectado à entrada não inversora do amplificador, e uma tensão de referência é gerada para a entrada inversora do conversor. Desta forma, caso a tensão de entrada seja superior à tensão de referência, teremos na saída do amplificador a tensão de VCC (alimentação) e, por outro lado, se a tensão de entrada for menor que a tensão de referência, veremos na saída do amplificador a tensão de GND, assim, podemos dizer que esse conversor é um conversor analógico-digital com resolução de 1 bit. Veja, na Figura 4.3, um exemplo de implementação deste conversor.

Figura 4.3 | Conversor analógico-digital de 1 bit: (a) esquemático de circuitos (b) onda resultante em Vout



(b)



Fonte: elaborada pela autora.

Neste exemplo, um sinal senoidal de entrada *Vin*, de amplitude de 5 V é aplicado na entrada não inversora do amplificador operacional, e na entrada inversora uma tensão de referência de 2,5 V. Sempre que a tensão de entrada é maior que *Vref*, a tensão de saída assume o valor lógico 1, nesse caso com uma tensão de 5V e, sempre que é menor, assume o valor de 0 V ou, lógico, 0.

Em situações em que é necessário mais do que uma saída (um bit) para cobrir várias faixas de valores de leitura, utilizam-se circuitos comparadores em paralelo e muda-se o valor da tensão de referência para cada um. Dessa forma, teremos uma quantidade de bits de saída proporcional à quantidade de comparadores, e a esse fator dá-se o nome de **Resolução**.

Reflita

Muitos dispositivos que atualmente operam com a leitura de sensores analógicos, dispensam a utilização de microcontroladores por necessitarem de uma baixa resolução de leitura do sinal. Um exemplo é o poste de luz, que possui um circuito que realiza a leitura de um sensor de luz (LDR) e compara esse valor a uma referência exata que define se está de dia ou de noite, de forma a acionar ou desligar a lâmpada. Quais outros dispositivos você conhece que operam de forma semelhante?

Amostragem

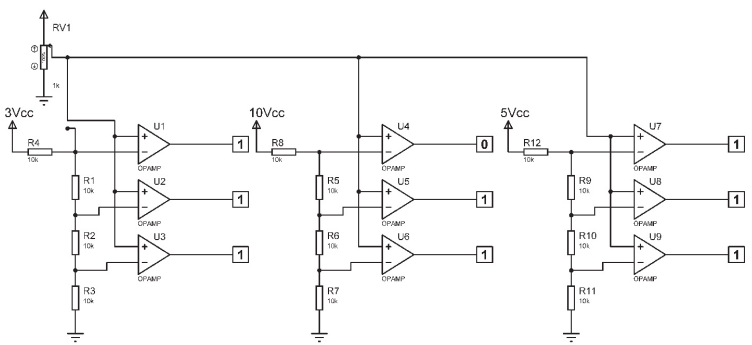
Outro fator muito importante para o desempenho dos conversores AD é a sua taxa de amostragem ou frequência de amostragem (do inglês frequency *sample*), ou seja, com qual velocidade o circuito será capaz de capturar um valor do sinal analógico de entrada, armazenar este valor, convertê-lo e finalmente atribuir o valor digital de saída, para só então estar novamente pronto para realizar o processo novamente.

A frequência de amostragem (fs) de um sinal analógico é independente da frequência do próprio sinal, porém, quanto maior ela for, mais informação teremos e, consequentemente, o microcontrolador será capaz de realizar o controle sobre seus atuadores de forma mais precisa.

Tensão de referência

Como pode ser observado no circuito apresentado na Figura 4.3, a tensão de referência (Vref) desempenha um papel crucial para o processo de conversão do sinal analógico para digital, pois é através dela que o sinal de entrada é comparado para se determinar qual deve ser o sinal digital de saída. Considere os três circuitos de conversores AD de 3 bits, apresentados na Figura 4.4:

Figura 4.4 | Conversores AD com diferentes tensões de referência



Fonte: elaborada pela autora.

Nessa simulação, foi aplicada uma tensão senoidal de 5 Vpp, e tensões de referência de 3 V, 5 V e 10 V, respectivamente, em cada circuito. No circuito 1, temos representações digitais proporcionais à

tensão analógica de entrada do valor de 0 V até 2,25 V e, a partir desse valor até o valor de 5 V, temos sempre o mesmo conjunto de saídas S = {1,1,1}. No circuito 2 (tensão de referência de 10 V), perceba que em nenhum momento a primeira saída será ativada, pois a tensão de referência para o seu comparador é de 7,5 V. E finalmente, temos o circuito com tensão de referência de 5 V, e nele podemos observar que as saídas são acionadas de maneira totalmente proporcional à entrada, pois sua tensão de referência coincide com o valor máximo do sinal. A seguir, uma tabela em resumo:

Tabela 4.1 | Conversores AD com diferentes tensões de referência

	Circuito 1 (Vref = 3V)			Circuito 2 (Vref = 10V)			Circuito 3 (Vref = 5V)		
Vin	Saída 1	Saída 2	Saída 3	Saída 1	Saída 2	Saída 3	Saída 1	Saída 2	Saída 3
0 V	0	0	0	0	0	0	0	0	0
0,5 V	0	0	0	0	0	0	0	0	0
1 V	0	0	1	0	0	0	0	0	0
1,5 V	0	1	1	0	0	0	0	0	1
2 V	0	1	1	0	0	0	0	0	1
2,5 V	1	1	1	0	0	1	0	1	1
3 V	1	1	1	0	0	1	0	1	1
3,5 V	1	1	1	0	0	1	0	1	1
4 V	1	1	1	0	0	1	1	1	1
4,5 V	1	1	1	0	0	1	1	1	1
5 V	1	1	1	0	1	1	1	1	1

Fonte: elaborada pela autora.

Podemos concluir que a escolha da tensão de referência deve ser tal que não ultrapasse o valor máximo que o sinal analógico poderá oferecer (Vpp), e que, da mesma forma, não seja muito acima desse mesmo valor, para que o conjunto de saídas digitais possa ser o mais simétrico e eficiente possível.

Conversor analógico-digital no ATmega328

O microcontrolador ATmega328 possui um periférico exclusivo para conversão de sinais analógicos para valores digitais. Este conversor possui as seguintes características principais:

- 10 bits máximos de resolução, resultando em até 1024 pontos de leitura (2^{10}).
- Tempo de conversão selecionável entre 13 µs e 260 µs
- 6 canais de leitura multiplexados.
- Tensão de referência selecionável.

- Interrupção ao final da conversão.
- Sensor interno de temperatura.

Apesar de o ATmega328 ser um microcontrolador com palavra de 8 bits, o seu conversor AD possui 10 bits máximos de resolução. Tal disparidade é resolvida pela separação do valor de leitura em dois registradores: ADCH e ADCL, em que as letras H e L significam High e Low, ou seja, parte alta e parte baixa da conversão. Obviamente, o processo de conversão se torna ligeiramente mais demorado devido a essa característica. Porém, quando em uma aplicação, não é necessário a utilização de uma resolução de 10 bits: é possível configurar o microcontrolador para realizar um conversor com 8 bits apenas, evitando, assim, o processo de separação dos dados. Isso é feito através do bit ADLAR do registrador ADMUX.

O bit ADSC do registrador ADCSRA possui dupla função, pois é através da sua ativação que se inicia o processo de conversão propriamente dito e, após isso, esse bit permanece em 1 até que o processo finalize. Ao término da conversão, o valor convertido é imediatamente transferido para o registrador ADC. É possível deduzir a tensão de entrada no canal utilizando a seguinte expressão:

$$V_{in} = \frac{ADC.V_{ref}}{2^{bits}}$$

Em que *V in* é a tensão analógica de entrada, ADC é o valor digital convertido, *V ref* é a tensão de referência, e bits é a quantidade de bits configurada como resolução do conversor (8 ou 10 bits).

Exemplificando

Qual seria a tensão de entrada de um sensor genérico, sabendo que o valor convertido pelo ATmega328 é de 413, sua tensão de referência é de 5 V, e sua resolução de 10 bits?

Aplicando a fórmula, temos:

$$V_{in} = \frac{ADC.V_{ref}}{2^{bits}} \rightarrow V_{in} = \frac{413x5,0}{2^{10}} \rightarrow V_{in} = \frac{413x5,0}{2^{10}} \rightarrow V_{in} = 2,02\ V$$

Modos de operação

O conversor AD do microcontrolador ATmega328 possui dois modos de operação: leitura única (*one-shot*) e conversão contínua (*free running*).

O primeiro modo de operação se dá através do processo descrito anteriormente, no qual o conversor é inicialmente configurado considerando sua tensão de referência e sua resolução (em bits), e após, é feita uma única leitura a cada acionamento do bit ADSC.

No modo de conversão contínua, ao término da leitura e conversão do valor de entrada, o microcontrolador automaticamente inicia uma nova conversão, dessa forma, o registrador ADC (ADCH e ADCL) sempre conterá o valor mais atualizado de conversão da entrada analógica.

Em ambos os modos de operação é possível habilitar uma interrupção dedicada para ser invocada sempre que ocorrer o término de uma conversão.

Tempo de conversão

O periférico ADC possui um *prescaler* interno controlado pelos bits ADPS0..2, que determina o período de amostragem do sinal. Esse valor deve ser determinado levando em consideração as características do sinal analisado, o tempo de resposta exigido pelo sistema para que um valor atualizado seja fornecido e também a qualidade da conversão, uma vez que períodos muito curtos de conversão podem comprometer a estabilidade e a confiabilidade do sinal que será adquirido pelo circuito.

Registradores ADMUX

Figura 4.5 | Registrador ADMUX

Bit		7	6	5	4	3	2	1	0
	ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0
Lê/Escreve		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

Fonte: Lima; Villaça (2012, p. 144).

- REFS1..0: selecionam a fonte de tensão de referência. {0,0} Pino AREF; {0,1} Pino AVCC; {1,1} Tensão interna de 1,1 V.

- ADLAR: 1 → Valor de conversão ajustado à esquerda (para resolução de 8 bits); 0 → Valor de conversão ajustado à direita (para resolução de 10 bits).
- MUX3..0: seleciona qual canal será conectado a ADC para conversão. {0,0,0,0} ADC0; {0,0,0,1} ADC1; {0,0,1,0} ADC2; {0,0,1,1} ADC3; {0,1,0,0} ADC4; {0,1,0,1} ADC5; {0,1,1,0} ADC6.

Assimile

É através do bit ADLAR do registrador ADMUX que definimos a resolução da nossa conversão. É válido destacar que conversões com menos bits são mais velozes, enquanto conversões com mais bits aumentam a precisão do sinal amostrado.

ADCSRA

Figura 4.6 | Registrador ADCSRA

Bit		7	6	5	4	3	2	1	0
	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Lê/Escreve		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

Fonte: Lima; Villaça (2012, p. 145).

- ADEN – Habilita o conversor AD.
- ADSC – Inicializa o processo de conversão.
- ADATE – Ativa o modo de autodisparo, em que o conversor é vinculado a outro periférico do controlador.
- ADIF – Ativo quando ocorre uma interrupção.
- ADIE – Habilita a chamada de uma interrupção ao término da conversão.
- ADPS2..0: seleção do fator de divisão (prescaler) em relação ao clock de entrada: {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.

ADCL / ADCH

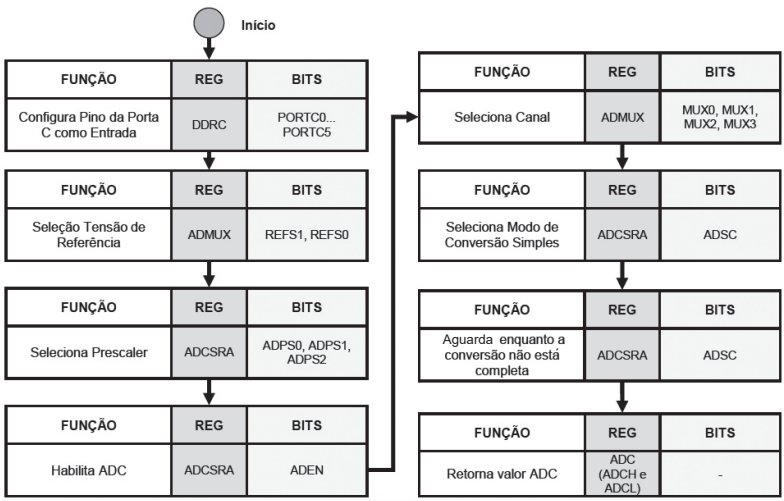
Ao término da conversão, o valor convertido é transferido para esses registradores. Caso o bit ADLAR seja configurado como 1

(resolução de 8 bits), basta realizar a leitura do registrado ADCL e descartar o valor de ADCH.

Resumo do fluxo de conversão simples

A seguir, o resumo do fluxo de uma conversão simples, e a relação de registrados com seus respectivos bits.

Figura 4.7 | Fluxo ADC - Conversão simples



Fonte: elaborada pelo autor.

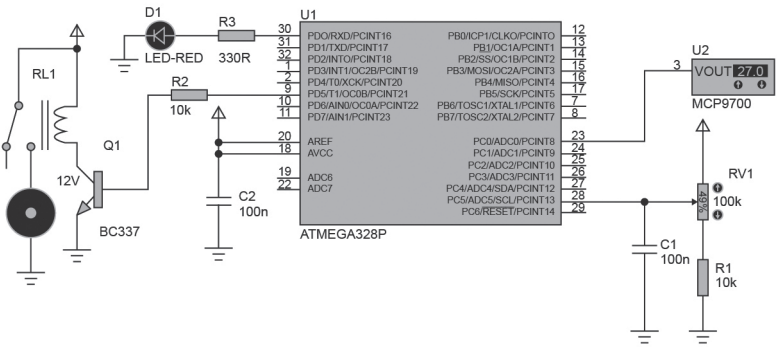
Sem medo de errar

Com o conhecimento adquirido nesta seção, você, como técnico responsável, está apto a realizar a implementação da nova especificação de produto solicitada pelo cliente da sua empresa. Neste projeto, lhe foi passado como requisito, que o sistema de controle de temperatura deve obrigatoriamente utilizar o sensor MCP9700, e também um potenciômetro, que será acoplado em um botão giratório para que o operador possa definir a temperatura de controle, que deve ser entre 25 °C e 100 °C. Para induzir o resfriamento e regular a temperatura do sistema a ser controlado, seu cliente também solicitou que uma ventoinha fosse utilizada no módulo de controle, e assim, através da comparação da temperatura atual e da configurada, o sistema a liga ou a desliga. Por fim, como último requisito, é solicitado que um LED

intermitente pisque em uma frequência proporcional à temperatura lida, como um meio de controle visual do usuário.

O primeiro passo para a resolução deste problema é definir as características dos nossos sinais de entrada e saída, e, com isso, configurar nossos registradores para que estejam os mais adequados possíveis para trabalhar dentro dos limites das nossas interfaces. Vamos iniciar por um possível diagrama de hardware que poderia ser utilizado para demonstração do problema:

Figura 4.8 | Diagrama de hardware



Fonte: elaborada pelo autor.

Como foi requisitado pelo cliente, o sistema deve realizar a leitura do sensor analógico MCP9700, cuja tensão de saída é diretamente proporcional à sua temperatura, em uma escala linear que vai de -50 °C (0 V) até 125 °C (1,8 V). Utilizaremos um potenciômetro que atuará como selecionador de temperatura, sabendo que a faixa de temperatura selecionável é de 25 °C até 100 °C, e que analisando o divisor resistivo formado entre ele e o resistor R1, temos uma variação de tensão entre 455 mV até 5 V, podemos então definir que: potenciômetro totalmente fechado (455 mV) = temperatura de 25 °C, e potenciômetro totalmente aberto (5 V) = temperatura de 100 °C. Considerando que utilizaremos 10 bits como resolução do nosso conversor, e que utilizaremos o pino AREF (5 V) como tensão de referência, então também podemos definir que ADC = 0 $\rightarrow$ 0 V, ADC = 1024 $\rightarrow$ 5 V. Assim, relacionado ao valor ADC como o valor de temperatura selecionada, chegamos à seguinte relação linear:

$$T_{set} = 0{,}08 x ADC + 17{,}5$$

Em que *T set* é a temperatura configurada pelo usuário e ADC é o valor lido entre 0 e 1024 do conversor AD.

Para a relação entre o valor ADC e a temperatura do sensor MCP9700, temos a seguinte função linear:

$$T_{sen} = 0{,}17xADC - 50$$

Em que *T sen* é o valor de temperatura lido pelo sensor.

Podemos utilizar o Timer 1 de 16 bits para realizar o pisca do LED, em que o valor de carregamento do seu registrador TCNT1 poderia ser 100 vezes o valor lido pelo ADC no canal 0 (sensor). Dessa forma, a frequência aumentaria gradativamente conforme o valor da temperatura também se elevasse.

A seguir, um possível código-fonte para solução do problema:

Figura 4.9 | Possível solução de software para o problema

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include <avr/interrupt.h>
#define set_bit(Reg,bit) (Reg|=(1<<bit))
#define reset_bit(Reg,bit) (Reg&=~(1<<bit))
#define troca_bit(Reg,bit) (Reg^=(1<<bit))
#define le_bit(Reg,bit) ((Reg>>bit)&0x01)

void hw_config(void);
unsigned int LeituraADC(unsigned char canal);
unsigned int leitura_sensor, leitura_pot;
unsigned int temp_sensor, temp_set;
volatile int carga_timer;

int main(){
 hw_config();
 while(1) {
  leitura_sensor = LeituraADC(0);
  leitura_pot = LeituraADC(CANAL_5);
  //Aplica a Equação de conversão
  temp_sensor = ((leitura_sensor * 8) / 100) + 17;
  temp_set = ((leitura_pot * 17) / 100) - 50;
  //Ajusta a frequência de pisca segundo a temperatura.
  carga_timer = leitura_sensor * 100;
  //Faz a comparação entre os valores
  if (temp_sensor >= temp_set) {
   set_bit(PORTD,PORTD5); // Liga Ventoinha
  }
  else{
   reset_bit(PORTD,PORTD5); // Desliga Ventoinha
  }
 }
}

void hw_config(){
 // Seleciono a Tensão de Referência como AVCC
 set_bit(ADMUX, REFS0);
 //Prescaler de 128
 set_bit(ADCSRA,ADPS2);
 set_bit(ADCSRA,ADPS1);
 set_bit(ADCSRA,ADPS0);
 // Habilito o Canal Analógico
 set_bit(ADCSRA, ADEN);
 DDRD = 0xFF;
 DDRC = 0x00;
 set_bit(DDRD, PORTD5);
 set_bit(DDRD, PORTD0);
 //Configura Timer 1 com prescaler de 1024
 set_bit(TCCR1B,CS12);
 set_bit(TCCR1B,CS10);
 set_bit(TIMSK1, TOIE1);
        sei();
}

unsigned int LeituraADC(unsigned char canal){
 //Seleciona o Canal AD, de forma segura
 ADMUX = (ADMUX & 0xF0) | (canal & 0x0F);
 //Conversão em Modo Simples
 set_bit(ADCSRA,ADSC);
 // Espera até a conversão estar completa
 while(le_bit(ADCSRA,ADSC));
 // Retorna o valor da conversão
 return ADC;
}

ISR(TIMER1_OVF_vect){
 troca_bit(PORTD, PORTD0);
 TCNT1 = carga_timer;
}
```

Fonte: elaborada pelo autor.

Avançando na prática

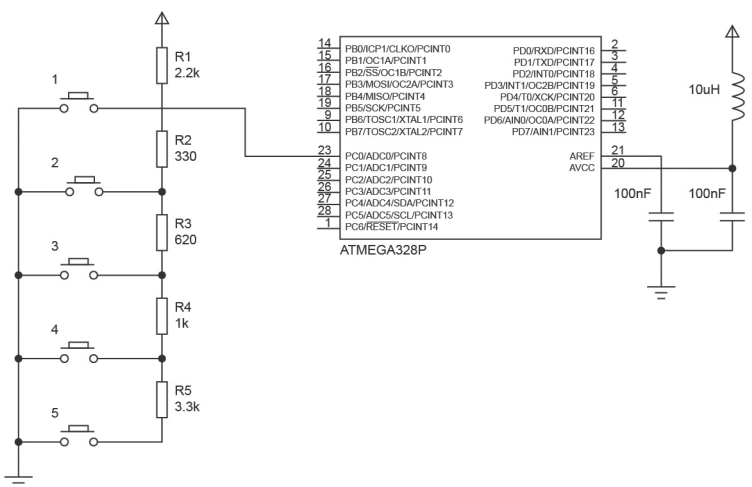
Teclado matricial analógico

Descrição da situação-problema

Assim como nos outros projetos em que você atuou como responsável técnico pelo desenvolvimento, nesse projeto de controlador de temperatura também foi especificado pelo cliente que o circuito deve prever um teclado simples com 5 teclas para seleção de um *preset* de temperatura que o microcontrolador deve considerar. Além disso, deve ser utilizado um display de 7 segmentos (semelhante ao visto em outras situações) para apresentar qual tecla foi pressionada de maneira numérica. No entanto, um imprevisto no projeto de hardware lhe trouxe a seguinte situação: não há disponível no microcontrolador um canal de I/O para cada tecla, então, você, com o seu conhecimento de conversores do tipo analógico-digital, propôs um circuito em que seria necessário somente um canal analógico para a leitura de todas as teclas.

Qual seria a sua proposta para a solução desse impasse, através da implementação de um software embarcado?

Figura 4.10 | Teclado analógico de 5 teclas



Fonte: Lima; Villaça (2012, p. 453).

Resolução da situação-problema

Para a solução do problema, deve-se considerar como serão feitas as leituras do teclado analógico, como serão comparadas e, por fim, como será acionado o display de 7 segmentos para a representação da tecla. Como demonstrado no diagrama da Figura 4.10, o canal analógico utilizado é o canal 0, logo, devemos preparar o pino PC0 como entrada e configurar o multiplexador do periférico AD para realizar a leitura desse canal. É possível notar que, a cada tecla pressionada, é formado um circuito divisor resistivo entre o resistor R1 e a soma resistiva dos resistores precedentes à tecla, que por consequência, resultará em uma tensão distinta em PC0. Por exemplo, ao se pressionar a tecla 4, temos um divisor resistivo entre R1 e a soma de R2, R3 e R4, e uma tensão resultante de 2,35 V PC0, que em valores AD será de 481. Deste modo, devemos, então, somente criar constantes que representam o limite AD de cada tecla para identificá-las. Por fim, para o acionamento do display de segmentos, podemos criar uma tabela com a combinação de bits para a representação de cada numeral, e assim, repassar esses valores à PORTA D.

Figura 4.11 | Possível solução de software para o problema

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#define set_bit(Reg,bit) (Reg|=(1<<bit))
#define reset_bit(Reg,bit) (Reg&=~(1<<bit))
#define troca_bit(Reg,bit) (Reg^=(1<<bit))
#define le_bit(Reg,bit) ((Reg>>bit)&0x01)

#define TECLA_5 0
#define TECLA_4 133
#define TECLA_3 307
#define TECLA_2 481
#define TECLA_1 717
#define MARGEM  20

void hw_config(void);
unsigned int LeituraADC(unsigned int canal);
unsigned int leitura;
unsigned char display[6] =
{0b11111111, 0b01101101, 0b01100110, 0b01001111,
0b01011011, 0b00000110 };

int main(){
 hw_config();
 while(1) {
  //Faz a leitura do canal analógico
  leitura = LeituraADC(0);
  //Checa qual tecla foi pressionada
  if (leitura <= TECLA_5 + MARGEM){PORTD = display[5];}
  else if (leitura <= TECLA_4 + MARGEM)  {PORTD = display[4];}
  else if (leitura <= TECLA_3 + MARGEM)  {PORTD = display[3];}
  else if (leitura <= TECLA_2 + MARGEM)  {PORTD = display[2];}
  else if (leitura <= TECLA_1 + MARGEM)  {PORTD = display[1];}
  else  {PORTD = display[0];}
 }
}
void hw_config()
{
// Seleciona a Tensão de Referência como AVCC
 set_bit(ADMUX, REFS0);
 //Prescaler de 128
 set_bit(ADCSRA,ADPS2);
 set_bit(ADCSRA,ADPS1);
 set_bit(ADCSRA,ADPS0);
 // Habilito o Canal Analógico
 set_bit(ADCSRA, ADEN);
 //Todos os canais de PORTD com saída
 DDRD = 0xFF;
 //Todos os canais de PORTC como entrada
 DDRC = 0x00;
}
unsigned int LeituraADC(unsigned int canal)
{
 //Seleciona o Canal AD, de forma segura
 ADMUX = (ADMUX & 0xF0) | (canal & 0x0F);
 //Conversão em Modo Simples
 set_bit(ADCSRA,ADSC);
 // Espera até a conversão estar completa
 while(le_bit(ADCSRA,ADSC));
 // Retorna o valor da conversão
 return ADC;
}
```

Fonte: elaborada pelo autor.

Faça valer a pena

1. Os conversores do tipo analógico-digital ou simplesmente ADC, possuem características que determinam a sua precisão e velocidade de conversão, chamadas de Resolução e Taxa de Amostragem, respectivamente. Esses fatores são muito importantes no momento da escolha de qual microcontrolador utilizar em virtude do sinal que deve ser lido e interpretado por seu conversor.
Considerando as características dos conversores do tipo analógico-digital qual dessas afirmativas pode ser considerada verdadeira?

a) Quanto maior a taxa de amostragem e menor a resolução, melhor será o conversor AD, uma vez que aumentará sua velocidade de conversão.
b) Quanto maior for a resolução, menor será a taxa de amostragem, pois assim existe uma relação inversa entre os dois fatores.
c) Não é possível obter altas resoluções com altas taxas de amostragem, pois um fator exerce uma influência obrigatória no outro.
d) Podemos afirmar que quanto maior a taxa de amostragem e maior a resolução, melhor será o conversor AD, uma vez que serão garantidos uma maior velocidade de conversão e valores de leitura mais precisos.
e) A velocidade de conversão e a precisão das leituras não são influenciadas pelo valor da resolução e da taxa de amostragem.

2. O primeiro passo para se configurar um periférico de conversão analógico-digital é analisar previamente o sinal que será mensurado, e avaliar algumas das suas características, como amplitude de sinal, taxa de variação, precisão mínima de leitura, dentre outros. Após este passo, a configuração do conversor deve atender aos limites impostos por esta análise.
Considere um sistema em que se deseja realizar a leitura de um sensor que possui uma variação do sinal entre 0,5 V e 3,2 V, e uma precisão de leitura de 100 mV é suficiente para a operação do sistema.
Assinale verdadeiro ou falso quanto à configuração do conversor AD do microcontrolador ATmega328:

I – Para aumentar o desempenho do conversor, este poderia ser configurado com uma resolução de 8 bits, uma vez que esse valor seria suficiente para a precisão solicitada.
II – A tensão de referência AREF deve ser selecionada para otimizar a precisão da leitura.
III – A tensão de referência deve ser de 0,5 V, uma vez que esse valor representa o início de escala do sinal.

IV – Deve ser selecionada a resolução de 10 bits para aumentar a resolução e a velocidade de conversão.
V – Somente uma conversão em modo de leitura simples pode ser configurada para operar com este sensor.

a) F, V, V, F, F.
b) F, F, V, V, F.
c) V, V, F, V, V.
d) V, V, F, F, F.
e) F, F, V, F, V.

3. Uma etapa muito importante do processo de conversão e análise de um sinal analógico é a interpretação da leitura e a sua possível equivalência à grandeza original lida pelo sensor de forma a sempre padronizar as unidades de medição.
Um sensor de temperatura é acoplado à entrada do canal 0 (PC0) do conversor analógico-digital. Esse sensor possui uma relação linear entre a temperatura e a sua tensão de saída, em termos que 10 °C representam uma tensão de saída de 4,5 V e 150 °C representam uma tensão de saída de 0,8 V.
Sabendo que o conversor AD do microcontrolador ATmega328 foi configurado com uma resolução de 10 bits e uma tensão de referência de 4,5 V, qual deve ser o valor real de temperatura medida pelo sensor quando o registrador ADC estiver com o valor de 345?

a) ~123 ºC.
b) ~45 ºC.
c) ~75 ºC.
d) ~15 ºC.
e) ~40 ºC.

Seção 4.2

Módulos de comunicação

Diálogo aberto

Vimos até aqui como o uso do microcontrolador torna os projetos eletrônicos mais versáteis e poderosos, uma vez que sendo considerado um computador em pequena escala, ele é capaz de controlar entradas e saídas, processar e converter dados, realizar a contagem de tempo, gerar sinais de onda, entre outras funcionalidades. Apesar de tamanhas vantagens, raramente um projeto avançado que necessite da sua utilização, contará somente com esse componente como meio de processamento de dados para sua operação, necessitando, dessa forma, que o microcontrolador e os demais dispositivos se comuniquem, a fim de trocar informações, sejam elas de entrada, saída ou armazenamento. Além desse aspecto, devido à sua inerente complexidade, é fundamental para alguns projetos que um canal de comunicação entre o microcontrolador e o mundo externo seja implementado, visando monitorar o seu funcionamento, coletar dados, atualizar seu software ou até mesmo modificar parâmetros de maneira on-line.

Nesta seção, entenderemos como o microcontrolador implementa alguns dos principais protocolos para comunicação entre dispositivos ou entre ele e outros sistemas externos, e quais propriedades devemos levar em consideração na hora de realizar a arquitetura do nosso projeto de eletrônica embarcada.

Você, como técnico responsável de uma empresa de controladores eletrônicos, foi designado para um projeto específico, que exige conhecimentos sobre protocolos de comunicação em microcontroladores. Foi requisitado pelo cliente da empresa o desenvolvimento de um sistema de monitoramento de temperatura que mandatoriamente utilize o sensor de temperatura digital TC72, que possui uma interface de comunicação SPI, permitindo que outro dispositivo colete a leitura do sensor via esse protocolo serial. Foi solicitada, também, a implementação de um terminal serial para acessar o valor lido de temperatura e realizar a configuração do sistema.

O microcontrolador deve realizar leituras de temperatura a cada 100 ms pelo sensor TC72 e armazenar os seguintes valores em sua memória RAM: a) a última leitura; b) o menor valor desde que o sistema foi energizado e c) o maior valor nas mesmas condições. Sempre que o sistema for inicializado, o microcontrolador deve considerar como temperatura configurada o valor de 50 °C, e se a temperatura atual for superior a esse valor, uma ventoinha deve ser acionada; caso contrário, desligada.

Pelo terminal serial, os seguintes comandos devem ser implementados: a) tecla 's': retorna o valor de temperatura atual, máxima, mínima e temperatura configurada; b) tecla 'v': retorna o estado da ventoinha; c) tecla 'u': aumenta o valor da temperatura configurada em 5 °C até o máximo de 120 °C e d) tecla 'd': diminui o valor da temperatura configurada em 5 °C até o mínimo de 25 °C. Pronto para tornar seus projetos com microcontroladores mais interativos? Então vamos lá!

Não pode faltar

Protocolos de comunicação

Como foi visto até agora, o microcontrolador é um dispositivo com inúmeros periféricos internos que lhe agrega flexibilidade e poder de computação necessários para controlar e interagir com os mais diversos componentes eletrônicos, seja por meio de interfaces analógicas ou digitais. No entanto, em muitos casos, é necessário que ele se comunique com outros sistemas, inclusive outros microcontroladores, para troca de informação, leitura ou escrita de dados, diagnose de funcionamento e armazenamento de valores. Diversos são os meios existentes para que essa comunicação ocorra e, para isso, esses processos são padronizados através de protocolos, que ditam como a informação deve trafegar entre dois ou mais dispositivos. A seguir, alguns exemplos desses protocolos:

- Transferência paralela simples: usada para transferir dados de 8,6,16,32... bits ao mesmo tempo.
- Transferência serial assíncrona (UART): um dos protocolos seriais mais antigos, porém, mais utilizados até os dias atuais, utilizando 2 vias de comunicação mais uma de terra (GND).

- Interface serial periférica (SPI): um modelo de comunicação muito utilizado para comunicação entre dois circuitos integrados.
- Universal Serial Bus (USB): um protocolo de comunicação serial muito avançado e com altas taxas de transferência, utilizado nos dias atuais na grande maioria dos computadores de forma a se conectar com quase todos os dispositivos eletrônicos mais modernos.

Pesquise mais

Inúmeros são os protocolos existentes atualmente, e sempre que uma nova tecnologia eletrônica emerge, novos protocolos podem ser criados para se adequar a ela. Para organizar todos esses protocolos, alguns modelos foram criados de forma a classificá-los segundo sua aplicação e abstração, dentre os principais está o Modelo de Camadas OSI, ou simplesmente Modelo OSI. Veja o vídeo a seguir para entender mais sobre os diversos tipos de protocolos e quais suas principais características: **Protocolos de comunicação de dados! O que são?** Disponível em: <https://www.youtube.com/watch?v=FVWa6n7U6t8>. Acesso em: 27 out. 2017.

Classificação

Quanto ao modo de comunicação, os protocolos podem ser divididos em dois grandes grupos:

- Transferência paralela: neste modo, um número fixo de bits (8, 16 etc.) são transferidos ao mesmo tempo. Desta forma, é necessário um barramento de comunicação com uma quantidade de vias correspondente à quantidade de bits. Este modo de comunicação é muito rápido, mas sua principal desvantagem é justamente a quantidade de linhas de comunicação necessária. Esse modo de comunicação é muito utilizado quando o dispositivo a se comunicar se encontra próximo ao microcontrolador ou processador, como é o caso de memórias RAM externas e os cartões PCI presentes dentro de um computador.
- Transferência serial: neste modo, somente um único bit é transferido por vez. Desta forma, se é necessário transferir 1 byte

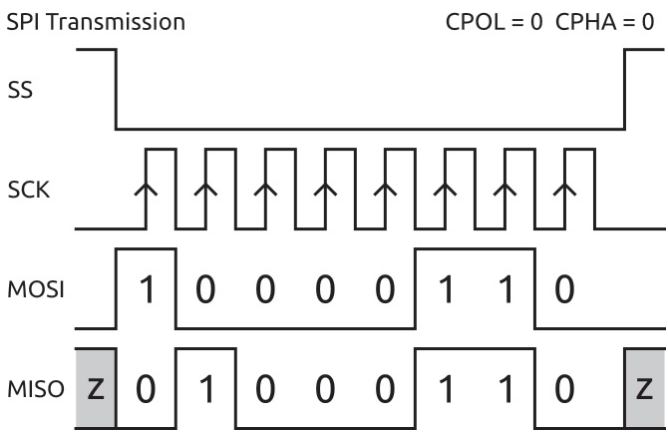
de dados, serão necessários oito ciclos de transferência. Apesar desta desvantagem, os protocolos seriais utilizam poucas vias físicas de comunicação para transferência de dados, o que significa uma economia em termos de custo e espaço.

Os modos de comunicação também podem ser classificados segundo a sincronicidade de transmissão, ou seja, se há ou não a necessidade de um sinal complementar para sincronizar ambos os dispositivos.

Transmissão síncrona

Neste tipo de transmissão, o pacote é enviado bit a bit pela linha da DADOS. A linha de *clock* sinaliza o fim do primeiro bit e o início do próximo. Quando a linha de *clock* muda seu valor (0 para 1 ou vice-versa) o bit é transferido, e assim, outro dispositivo pode lê-lo e armazená-lo. Veja na Figura 4.12 um exemplo da transferência de 1 byte com os valores 100001100, considerando o valor de borda de subida (0 → 1) da linha de *clock* utilizando um periférico SPI.

Figura 4.12 | Transferência síncrona SPI do valor 1000110



Fonte: elaborada pelo autor.

Transmissão assíncrona

Uma transmissão assíncrona permite que os dados sejam enviados sem que uma das partes tenha que enviar um sinal de *clock*. Para que a comunicação ocorra, ambos os dispositivos devem ter seus tempos

de transmissão e recepção alinhados e configurados previamente, e alguns bits especiais são incluídos no pacote para sincronizá-los. Um exemplo é o *Start Bit* (bit de inicialização), que, adicionado ao início de cada pacote, é utilizado para alertar o outro dispositivo, de forma que se prepare para o recebimento dos dados.

Taxa de transmissão ou *baud rate*

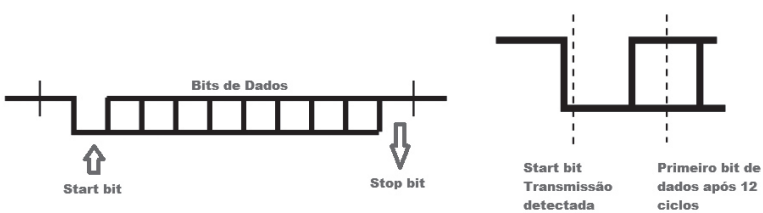
Baud rate é a medida padrão para velocidade de transmissão em sistemas assíncronos, dada em bits por segundo (bps). Todos os dispositivos em uma rede assíncrona devem possuir o mesmo valor de *baud rate* para que não haja falha de comunicação.

Pacote de dados em transmissão assíncrona

Um pacote de dados em uma comunicação assíncrona pode conter alguns bits de controle para garantir a entrega confiável da informação entre os dispositivos. Dentre os principais, podemos destacar:

- Bit de paridade: bit complementar adicionado ao final do pacote como um meio de análise simples de falha. Em um pacote com paridade par, o bit de paridade será 0 se a quantidade de bits '1' no pacote de dados for *par*; em um pacote com paridade *ímpar*, a lógica é inversa. O dispositivo que recebe o pacote pode checar a consistência dos dados recebidos, recalculando a paridade do pacote e confrontando com o bit recebido de paridade.
- Bit de parada ou *Stop bit*: bit que indica o final do pacote de dados.

Figura 4.13 | Exemplo de pacote de dados em comunicação assíncrona



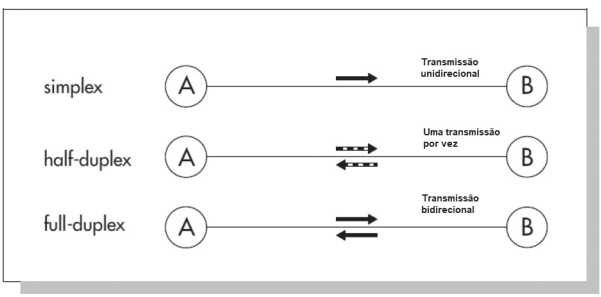
Fonte: elaborada pelo autor.

Técnicas de comunicação

Por fim, podemos classificar um protocolo de comunicação segundo as direções de comunicação entre os dispositivos:

- *Simplex*: transmissão unidirecional de um dispositivo para outro. Exemplos: transmissão de TV e rádio.
- *Half-duplex*: transmissão bidirecional; porém, só pode ocorrer uma transferência de dados por vez. Exemplo: rádios de comunicação (walkie-talkie).
- *Full-duplex:* transmissão bidirecional em que as transferências de dados podem ocorrer em ambas as vias entre os dispositivos, de maneira paralela. Exemplo: aparelhos celulares.

Figura 4.14 | Técnicas de comunicação



Fonte: elaborada pelo autor.

Reflita

Analisando as técnicas de comunicação, fica evidente que os dispositivos com comunicação *full-duplex* apresentam vantagens em relação aos demais dispositivos em termos de transferência de dados. Então, quais seriam as vantagens que poderiam ser destacadas para as demais técnicas? Por que ainda encontramos tantos dispositivos operando em modo *Half-duplex* ou até mesmo em *simplex*?

Periférico USART ATmega328

O periférico USART (*Universal Synchronous and Asynchronous Serial Receiver and Transmitter*) é um modulo de comunicação serial com inúmeras possibilidades de configurações de trabalho, o que lhe permite ser aplicado em uma infinidade de sistemas eletrônicos,

como nas comunicações RS-232 e RS-485, 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 (LIMA; VILLAÇA, 2012).

As principais características do periférico USART do microcontrolador ATmega328 são:

- Modo de transmissão *Full-Duplex*, com um registrador de recepção e outro de transmissão.
- Modo de operação síncrono ou assíncrono.
- Gerador de *baud rate* de alta resolução.
- Pode utilizar paridade par ou ímpar, além de realizar a conferência de paridade por hardware.
- Detecção de colisão de dados e erros de pacotes.
- Três fontes de interrupção distintas: transmissão completa, recepção completa e registrador de dados vazio.
- Pode ser utilizado como interface SPI.

O periférico USART possui um contador muito semelhante àqueles estudados na Unidade 3. Este contador é conectado diretamente ao *clock* do microcontrolador e utiliza o registrador UBRR0 para determinar o valor de contagem, sendo o responsável pela geração da taxa de transmissão (*baud rate*). Ele ainda pode utilizar um divisor (semelhante ao *prescaler* dos temporizadores) para dividir o valor do *clock* de entrada e conseguir valores mais exatos de taxa de transmissão. Na tabela a seguir, são apresentadas as fórmulas utilizadas para se calcular a taxa de comunicação segundo a configuração do registrador UBRR0, e também a relação inversa.

Tabela 4.2 | Equações para cálculo da taxa de transmissão e valor do registrador UBRR0

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	$TAXA = \frac{f_{osc}}{2(UBRR0 + 1)}$	$UBRR0 = \frac{f_{osc}}{2.TAXA} - 1$

Fonte: Lima; Villaça (2012, p. 346).

Apesar de periférico, pode ser configurado para operar tanto como módulo síncrono, assíncrono e SPI. Focaremos nossa atenção para o modo assíncrono, que engloba a parcela mais significante das aplicações com esse módulo.

Pesquise mais

Entenda mais sobre quais são os periféricos utilizados pelo periférico USART no modo de transmissão assíncrona e como podemos configurá-los: **Uso dos registros UCSRA, UCSRB, UCSRC, UDR, UBRRL e UBRRH na comunicação assíncrona.** Disponível em: <http://www.arnerobotics.com.br/eletronica/Configurando_corretamente_regsAVR_pt5.htm>. Acesso em: 27 out. 2017.

O pacote de dados que será enviado ou recebido pelo módulo USART em modo assíncrono pode ser configurado da seguinte maneira:

- Um bit de início (*start bit*).
- 5 a 9 bits de dados.
- Um bit de paridade par, ímpar ou nenhum.
- Um ou dois bits de parada (*stop bit*).

Na comunicação do módulo USART, inicialmente é enviado o bit de início, depois os bits de dados (iniciando pelo LSB); depois, é inserido o bit de paridade (caso seja configurado), e por fim, temos o bit de parada.

Transmissão de dados pela USART

A transmissão é iniciada quando acessamos o registrador UDR0 e escrevemos o dado a ser enviado. Se o módulo está vazio (*idle*), ou seja, não possui nenhum bit esperando para ser transmitido, então o valor de UDR0 é enviado para a máquina de estados do módulo, que preparará o pacote para ser enviado, e neste momento, o bit UDRE0 vai a nível lógico 1, indicando que a USART está pronta para receber o próximo pacote. Ao final da transmissão do pacote enviado a UDR0, o bit TXC0 do registrador UCSR0A vai a nível lógico 1, indicando que a transmissão foi concluída.

Como o pino TXD é multiplexado com o canal PD1 da porta D, devemos também configurar o bit TXEN0 como 1, no registrador UCSR0B, para que possamos habilitar essa função no pino.

Recepção de dados pela USART

O módulo USART inicia a recepção do pacote de dados, quando um bit de início válido é detectado. Os bits seguintes são armazenados no registrador UDR0 e o bit RXC0 do registrador UCSR0A vai a nível 1, indicando que o dado está pronto para ser lido pelo programa. Da mesma forma que o pino TXD, o pino RXD também precisa ser habilitado, pois é multiplexado com o canal PD0. Isso é feito elevando a nível 1 o bit RXEN0 do registrador UCSR0A.

Exemplificando

Neste exemplo é demonstrado como projetar uma configuração simples do periférico USART para receber e enviar um caractere. Nesta aplicação, é aguardado o recebimento de um caractere numérico, caso este esteja entre o valor 0x30 ('0' em ASCII) e 0x38 ('8' em ASCII), o valor é incrementado e enviado de volta ao transmissor; caso contrário, o microcontrolador envia de volta o caractere recebido.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#define set_bit(Reg,bit) (Reg|=(1<<bit))
#define reset_bit(Reg,bit) (Reg&=~(1<<bit))
#define troca_bit(Reg,bit) (Reg^=(1<<bit))
#define le_bit(Reg,bit) ((Reg>>bit)&0x01)

#define TAXA_TRANS  9600
#define MYUBRR (F_CPU/16/TAXA_TRANS-1)

unsigned char USART_le(void);
void USART_escreve(unsigned char caracter);
void USART_config(unsigned int usart);
unsigned char dado_recebido, dado_envia;

int main(){
  USART_config(MYUBRR);
  while(1)
  {
    dado_recebido = USART_le();
    if ((dado_recebido >= 0x30) &&
(dado_recebido   <= 0x38))
    {
      dado_envia = dado_recebido + 1;
      USART_escreve(dado_envia);
    }
    else {USART_escreve(dado_recebido);}
  }
}
```

```
void USART_config(unsigned int usart)
{
  //Ajusta a taxa de transmissão
  UBRR0H = (unsigned char)(MYUBRR>>8);
  UBRR0L = (unsigned char)MYUBRR;
  //Habilita os pinos de TXD e RXD
  UCSR0B = (1<<RXEN0)|(1<<TXEN0);
  /*Modo assíncrono, 8 bits de dados, sem
paridade*/
  UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
}
unsigned char USART_le(void)
{
  /*Aguarda o dado ser recebido e
retorna o valor*/
  while (!(UCSR0A & (1<<RXC0)));
  return UDR0;
}

void USART_escreve(unsigned char caracter)
{
  //Aguarda o dado ser enviado
  while (!( UCSR0A & (1<<UDRE0)) );
  UDR0 = caracter;
}
```

Fonte: elaborado pelo autor.

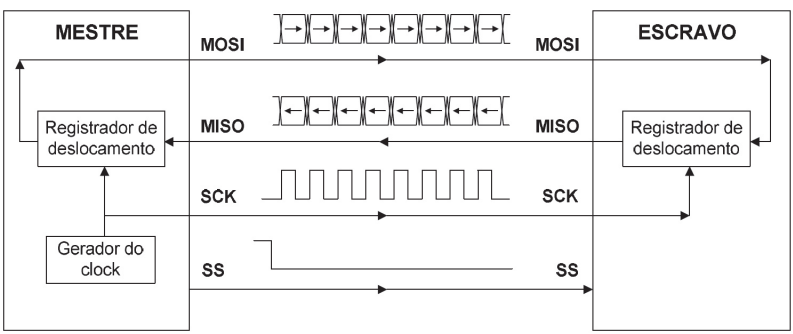
Periférico SPI para o microcontrolador ATMega328

Dentre os protocolos de comunicação serial síncrona, sem dúvidas, o SPI (*Serial Peripheral Interface*) é um dos mais utilizados para a comunicação entre o microcontrolador e diversos outros dispositivos que utilizam o mesmo protocolo, como sensores, controladores digitais, memórias, entre outros.

O protocolo SPI foi criado pela empresa *Motorola*, e trabalha no modo *full duplex*, podendo transmitir pacotes de dados de quaisquer tamanhos, com altas velocidades de comunicação. Ele se baseia em 4 sinais: a) MOSI: saída de dados do mestre e entrada do escravo; b) MISO: inverso do MOSI; c) SCK: *clock* gerado pelo mestre e d) SS: seleção de escravo.

O mestre é responsável por iniciar a comunicação através do pino SS, que seleciona o dispositivo escravo que vai receber o pacote. Após, ele inicia a geração os pulsos de *clock*, em que em cada pulso ocorrerá a transferência de um bit de dados do mestre para o escravo e vice-versa.

Figura 4.15 | Conexão Mestre-Escravo no protocolo SPI



Fonte: Lima; Villaça (2012, p. 346).

Assimile

Lembre-se de que, diferentemente dos protocolos assíncronos como a USART, o protocolo síncrono depende que um dos agentes gere um sinal de *clock*, de forma que os dispositivos se sincronizem no envio dos bits de dados.

Após o término da transmissão, o *clock* do SPI é interrompido e o bit SPIF do registrador SPSR vai a 1, e se o bit SPIE do mesmo registrador estiver em 1, uma interrupção é gerada para indicar o final do processo.

Registradores SPI

Figura 4.16 | Registrador ATmega328 para o periférico SPI

Fonte: elaborada pelo autor.

O registrador SPCR contém os bits responsáveis pela configuração do periférico SPI, e são eles:

- SPIE: habilita a interrupção do periférico.
- SPE: habilita o SPI.
- DORD: em 1 transmite-se primeiro o LSB, em 0 transmite-se o MSB.
- MSTR: em 1 configura o SPI como mestre, em 0 como escravo.
- CPOL: define a polaridade do *clock* quando inativo. Em 1, mantém-se em nível alto, em 0, mantém-se em nível baixo.
- CPHA: esse bit define se o dado será coletado na subida (1) ou descida (0) do *clock*.
- SPR1..0: esses bits controlam a taxa do sinal de *clock*, sendo {0,0} → fosc/4, {0,1} → fosc/16, {1,0} → fosc/64 e {1,1} → fosc/128.
- O registrador SPSR contém os bits de estado de operação do SPI, sendo eles:
- SPIF: vai a 1 quando uma transferência é finalizada.
- WCOL: é ativo indicando uma colisão, ou seja, quando se tenta escrever no registrador SPDR durante uma transmissão.
- SPI2X: quando em 1, dobra a velocidade configurada pelos bits SPR1..0 de SPCR.

O registrador SPDR é utilizado para enviar os dados em uma transmissão, ou para lê-los quando temos a recepção de um pacote.

Exemplificando

Nestes exemplos são demonstradas quais rotinas poderiam ser criadas para operar o periférico SPI como mestre ou como escravo.

```
/*Rotinas de SPI como Mestre*/
void SPI_Mestre_config(){
  DDRB = (1<<PB5)|(1<<PB3); //MOSI e SCK com saída
  //Habilita SPI como Mestre e clock de f_osc/16
  SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}
void SPI_Mestre_envia(char pacote){
  SPDR = pacote; // Inicia a transmissão
  while(!(SPSR & (1<<SPIF))); //Aguarda finalizar
}


/*Rotinas de SPI como Escravo*/
void SPI_Escravo_config(){
  DDRB = (1<<PB4);//MISO como saída
  //Habilita SPI como Mestre e clock de f_osc/16
  SPCR = (1<<SPE);//Habilita SPI
}
char SPI_Escravo_recebe(char p){
  while(!(SPSR & (1<<SPIF)));//espera recepção
  return SPDR; //retorna o valor recebido
}
```

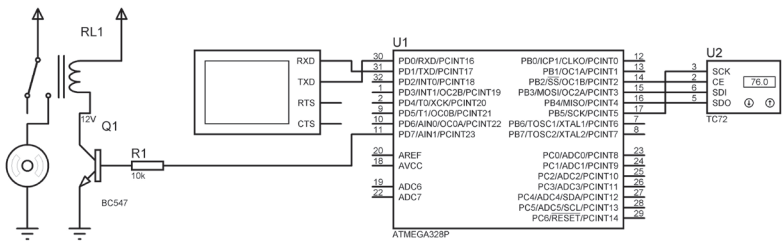
Fonte: elaborada pelo autor.

Sem medo de errar

Vamos, agora, resolver a situação-problema que foi proposta, mas antes, vamos relembrar o que foi solicitado. Você, como engenheiro responsável pelo projeto de eletrônica embarcada da sua empresa, foi designado para desenvolver um projeto de um controlador de temperatura interativo utilizando o circuito integrado TC72 e também um canal de comunicação serial para configuração e monitoramento do sistema. Na requisição do projeto, foi solicitado que, além de se utilizar o C.I. TC72, o projeto tenha alguns comandos programados para ler alguns valores de temperatura medidos (tecla 's'), ler o estado da ventoinha ('v') e também configurar pelo terminal a temperatura de controle (teclas 'u' para incrementar e 'd' para decrementar a temperatura).

O primeiro passo para a resolução desse problema é projetar o circuito eletrônico para embarcar o software de controle. Vale ressaltar que é fundamental que os componentes TC72 sejam conectados corretamente aos pinos de SPI do microcontrolador ATmega328, e que o terminal seja conectado aos respectivos pinos RXD e TXD.

Figura 4.17 | Diagrama de hardware



Fonte: elaborada pelo autor.

No software, temos que considerar a configuração do microcontrolador ATmega328 com o módulo SPI atuando como mestre na comunicação com o C.I. TC72. Segundo o manual dos componentes, para acessarmos o valor atualizado de temperatura, devemos primeiramente enviar o comando 0x02, e logo após enviar dois comandos 0x00, sendo que, no primeiro comando, o sensor retornará o valor inteiro da temperatura atual e no segundo, o sensor retornará o valor decimal da temperatura.

Na configuração do software para o terminal de acesso serial, temos que ler a todo momento o registrador de dados do canal USART e checar se algum dos comandos requisitados foi recebido e, em caso positivo, devemos utilizar o mesmo canal serial para responder adequadamente com os valores correspondentes ao comando.

O controle da ventoinha permanece muito parecido ao problema descrito na seção anterior, ou seja, basta compararmos sempre o valor atualizado do sensor com o valor configurado de temperatura, para determinarmos quando ela deve ser acionada.

Figura 4.18 | Proposta de software para a solução do problema

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include <stdio.h>

#define set_bit(Reg,bit) (Reg|=(1<<bit))
#define reset_bit(Reg,bit) (Reg&=~(1<<bit))
#define troca_bit(Reg,bit) (Reg^=(1<<bit))
#define le_bit(Reg,bit) ((Reg>>bit)&0x01)

#define VENT_ON()     set_bit(PORTD, PD7)
#define VENT_OFF()    reset_bit(PORTD, PD7)
#define TC72_ON()     set_bit(PORTB,PB2)
#define TC72_OFF()    reset_bit(PORTB,PB2)

#define TAXA_TRANS  19200
#define MYUBRR (F_CPU/16/TAXA_TRANS-1)

unsigned char USART_le(void);
void SPI_Mestre_config();
unsigned char SPI(unsigned char pacote);
unsigned char USART_le(void);
void USART_escreve(unsigned char caracter);
void USART_escreve(unsigned char caracter);
void USART_config(unsigned int usart);

unsigned char dado_recebido, dado_envia;
unsigned char tempInt, tempDec, MaxTemp, MinTemp;
unsigned char TempControl = 50;
unsigned char comando;
unsigned char vent;
char string[60] = {0x00};

int main(void)
{
  int i = 0;
  USART_config(MYUBRR);
  SPI_Mestre_config();
  set_bit(DDRD, PD7);
  TC72_ON();
  SPI(0x80);
  SPI(0x00);
  TC72_OFF();

    while (1)
    {
      TC72_ON();
      SPI(0x02);
      tempInt = SPI(0x00);
      tempDec = SPI(0x00);
      TC72_OFF();
      if(tempInt < MinTemp) {MinTemp = tempInt;}
      if(tempInt > MaxTemp) {MaxTemp = tempInt;}
      if(tempInt > TempControl) {VENT_ON(); vent = 1;} else
{VENT_OFF(); vent = 0;}
      comando = USART_le();
      switch(comando)
      {
        case 's':
          sprintf(string, "T. Atual: %dC T.Max: %dC T.Min: %dC
T.Set: %dC \r\n",tempInt,MaxTemp,MinTemp,TempControl);
          while(string[i] != 0x00) {USART_escreve(string[i]); i++;}
        break;
        case 'v':
          vent == 1 ? sprintf(string, "Ventoinha ligada! \r\n") :
sprintf(string, "Ventoinha desligada! \r\n");
          while(string[i] != 0x00) {USART_escreve(string[i]); i++;}
        break;
        case 'u':
          if(TempControl == 120) {TempControl = 120;} else
{TempControl += 5;}
        break;
        case 'd':
          if(TempControl == 25) {TempControl = 25;} else
{TempControl -= 5;}
        break;
        }
        string[0] = 0x00;
        i = 0;
    }
}

void SPI_Mestre_config(){
  DDRB = (1<<PB5)|(1<<PB3)|(1<<PB2);
  SPCR = (1<<SPE)|(1<<MSTR)|(SPR1)|(1<<CPHA);
}
unsigned char SPI(unsigned char pacote){
  SPDR = pacote;
  while(!(SPSR & (1<<SPIF)));
  return SPDR;
}

void USART_config(unsigned int usart)
{
  UBRR0H = (unsigned char)(MYUBRR>>8);
  UBRR0L = (unsigned char)MYUBRR;
  UCSR0B = (1<<RXEN0)|(1<<TXEN0);
  UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
}

unsigned char USART_le(void)
{
  if(UCSR0A & (1<<RXC0)) {return UDR0;} else {return 0x00;}
}

void USART_escreve(unsigned char caracter)
{
  while (!( UCSR0A & (1<<UDRE0)));
  UDR0 = caracter;
}
```

Fonte: elaborada pelo autor.

Avançando na prática

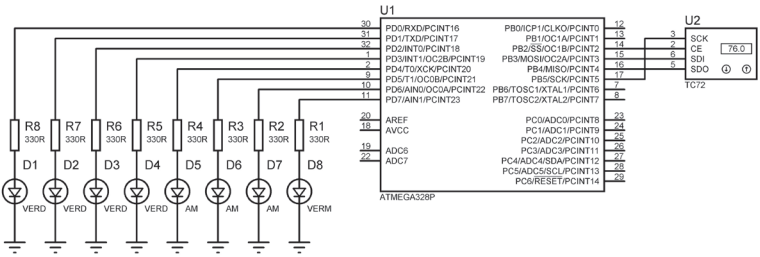
Indicador luminoso de temperatura

Descrição da situação-problema

Neste novo projeto, um cliente da empresa em que você trabalha solicitou que você projetasse um sistema de controle visual de temperatura, em que devem ser utilizados 8 LEDs para indicar o valor de temperatura lido por um sensor do tipo TC72. O cliente passou como especificação que o valor mínimo para a leitura de primeiro LED deve ser de 30 °C, e o valor máximo para o acendimento de todos os LED deve ser de 100 °C (10 °C para cada LED), sendo que os primeiros 4 LEDs devem possuir a cor verde, os próximos 3, a cor amarela, e o último, a cor vermelha.

Um colega engenheiro, que é responsável pelos projetos de hardware da empresa, já elaborou o circuito que será entregue ao cliente, e você, como engenheiro de software embarcado, foi escalado para o desenvolvimento do software que será embarcado no microcontrolador ATmega328, que embarca a placa eletrônica desse projeto. Qual seria uma possível solução de software para a resolução desse projeto?

Figura 4.19 | Diagrama de Hardware - indicador luminoso



Fonte: elaborada pelo autor.

Resolução da situação-problema

A solução para o problema passa pela leitura via módulo SPI do sensor de temperatura após definir para cada faixa de temperatura (partindo de 30 °C até 100 °C) quais os canais da porta D devem ser acionados segundo os LEDs, que por eles são controlados.

Figura 4.20 | Possível resolução de software para o indicador luminoso

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>

#define set_bit(Reg,bit) (Reg|=(1<<bit))
#define reset_bit(Reg,bit) (Reg&=~(1<<bit))
#define troca_bit(Reg,bit) (Reg^=(1<<bit))
#define le_bit(Reg,bit) ((Reg>>bit)&0x01)

#define TC72_ON()      set_bit(PORTB,PB2)
#define TC72_OFF()     reset_bit(PORTB,PB2)

unsigned char USART_le(void);
void SPI_Mestre_config();
unsigned char SPI(unsigned char pacote);

unsigned char tempInt, tempDec;
unsigned char tabelaLeds[8] =
{0b00000001,
 0b00000011,
 0b00000111,
 0b00001111,
 0b00011111,
 0b00111111,
 0b01111111,
 0b11111111};

int main(void)
{
  SPI_Mestre_config();
  DDRD = 0xFF;
  TC72_ON();
  SPI(0x80);
  SPI(0x00);
  TC72_OFF();
  while (1)
  {
    TC72_ON();
    SPI(0x02);
    tempInt = SPI(0x00);
    tempDec = SPI(0x00);
    TC72_OFF();

    if((tempInt >= 30) && (tempInt < 40)) {PORTD =
tabelaLeds[0];}
     else if((tempInt >= 40) && (tempInt < 50)) {PORTD =
tabelaLeds[1];}
     else if((tempInt >= 50) && (tempInt < 60)) {PORTD =
tabelaLeds[2];}
     else if((tempInt >= 60) && (tempInt < 70)) {PORTD =
tabelaLeds[3];}
     else if((tempInt >= 70) && (tempInt < 80)) {PORTD =
tabelaLeds[4];}
     else if((tempInt >= 80) && (tempInt < 90)) {PORTD =
tabelaLeds[5];}
     else if((tempInt >= 90) && (tempInt < 100)) {PORTD =
tabelaLeds[6];}
     else if(tempInt >= 100) {PORTD = tabelaLeds[7];}
     else {PORTD = 0x00;}
     }
}

void SPI_Mestre_config(){
 DDRB = (1<<PB5)|(1<<PB3)|(1<<PB2);
  SPCR = (1<<SPE)|(1<<MSTR)|(SPR1)|(1<<CPHA);
}

unsigned char SPI(unsigned char pacote){
  SPDR = pacote;
  while(!(SPSR & (1<<SPIF)));
  return SPDR;
}
```

Fonte: elaborada pelo autor.

Faça valer a pena

1. Com relação ao modo de transmissão, podemos classificar os protocolos de comunicação como síncronos e assíncronos, sendo que a forma como os dados são transmitidos e a quantidade de vias de comunicação diferem entre eles, concedendo-lhes vantagens e desvantagens de acordo com a sua aplicação.
Sobre os modos de transmissão síncronos e assíncronos, qual das afirmações é verdadeira?

a) Uma das premissas dos protocolos assíncronos é que um dos dispositivos deve gerar um sinal de *clock* que será utilizado por ambos para transmissão dos pacotes.
b) Alguns exemplos de protocolos síncronos são: SPI, RS232 e I2C.
c) Apesar de diferirem um do outro, os tipos dos sinais utilizados em ambos os módulos são os mesmos.
d) Utilizando o protocolo assíncrono, é fundamental que os dispositivos tenham seus parâmetros alinhados, uma vez que não haverá um sinal de sincronismo.
e) É comumente utilizado um bit inicial (*Start bit*) nos pacotes de protocolos síncronos, para que seja sinalizado o início do envio de pacotes.

2. O periférico USART é um módulo presente na grande maioria dos microcontroladores modernos, sendo flexível o suficiente para a implementação de protocolos assíncronos, como RS242, RS485 ou protocolos síncronos, como SPI.
Em qual das aplicações a seguir não é recomendável o uso do periférico USART para implementação do protocolo de comunicação?

a) Comunicação serial entre um microcontrolador e um computador para monitoramento do sistema embarcado.
b) Comunicação entre um microcontrolador e uma memória externa por um barramento paralelo.
c) Comunicação entre dois microcontroladores, sendo um escravo e outro mestre.
d) Implementação de um protocolo para configuração dos parâmetros do microcontrolador remotamente.
e) Projeto de um sistema *half-duplex* para comunicação via rádio.

3. O periférico SPI é muito utilizado para comunicação em microcontroladores ou microprocessadores e dispositivos eletrônicos diversos, como sensores, memórias, componentes de controle etc. Sua arquitetura essencialmente simples e confiável permite que mesmo os sistemas com baixo poder de processamento e memória possam implementar aplicações de comunicação sofisticadas e velozes.
Sobre o módulo SPI no microcontrolador ATMega328, quais das afirmações a seguir são verdadeiras e quais são falsas?

I- O protocolo SPI pode ser configurado tanto em modo assíncrono como síncrono.
II- O modulo SPI pode ser configurado para atuar como mestre em uma comunicação serial ou escravo, mas nunca em ambos ao mesmo tempo.
III- É possível dobrar a velocidade de comunicação do periférico, pela simples configuração de um bit específico.
IV- Os sinais utilizados para implementação do protocolo são: RXD, TXD, *Clock* e SS.

a) V, V, V, F.
b) F, V, F, V.
c) F, F, V, V.
d) V, F, V, V.
e) F, V, V, F.

Seção 4.3

Memória EEPROM

Diálogo aberto

Estamos chegando à reta final da nossa jornada em busca do conhecimento sobre microcontroladores e eletrônica embarcada. Já desbravamos as principais características de um sistema microcontrolado, os seus periféricos, as formas de programação e como utilizar a arquitetura de um microcontrolador a nosso favor na hora de desenvolver um projeto eletrônico.

Os projetos eletrônicos atuais, permeados de complexidade e sujeitos a condições adversas, devem apresentar alta robustez e confiabilidade no tratamento de dados e na sua operabilidade. Em muitos casos, não é possível ter o mapeamento completo das condições e varáveis a que o microcontrolador será submetido, como é o caso da falta inesperada de alimentação do circuito, em que todos os dados presentes na memória RAM são perdidos, incluindo leituras de sensores e estados internos do software. Para contornar esse problema, temos a nosso favor, um importante periférico: a memória EEPROM. Através dela, podemos armazenar os dados que são críticos para a nossa aplicação e que não queremos que sejam perdidos quando houver cerceamento da alimentação ou em condições adversas que possam resultar, inclusive, na reinicialização do microcontrolador.

Após o sucesso de seu último projeto como responsável técnico, sua empresa decidiu lhe transferir para o departamento de desenvolvimento de sistemas críticos, sendo que nesse departamento são desenvolvidas as rotinas de software que requerem maior atenção e que são responsáveis por garantir a robustez do sistema como um todo.

A sua primeira tarefa nesse novo departamento é a de justamente atualizar o último projeto desenvolvido, registrando na memória EEPROM do microcontrolador os valores computados e também a configuração de temperatura realizada pelo usuário. Assim, essas são as especificações para este projeto:

Utilizar o sensor digital de temperatura TC72 para realizar leituras a cada 100 ms e armazenar os seguintes valores em memória RAM: a) a última leitura; b) o menor valor desde que o sistema foi energizado e c) o maior valor nas mesmas condições. Dessa vez, esses mesmos valores com o estado da ventoinha devem ser armazenados em memória EEPROM, só que em uma frequência menor, a cada 10 s. Pelo terminal serial, temos os seguintes comandos de tecla: a) 's': retorna o valor de temperatura atual, máxima, mínima e temperatura configurada; b) 'v': retorna o estado da ventoinha; c) 'u': aumenta o valor da temperatura configurada em 5 °C até o máximo de 120 °C, e salva em EEPROM e d) 'd': diminui o valor da temperatura configurada em 5 °C até o mínimo de 25 °C, e também salva em EEPROM. Será adicionado mais um comando de tecla: 'z', para zerar todos os valores da memória EEPROM. Nesse novo cenário, sempre que o sistema for reiniciado, o microcontrolador deve buscar da memória EEPROM os valores de temperatura mínima, máxima e configurada. Está pronto para o seu projeto mais desafiador?

Não pode faltar

Principais tipos de memória

Iniciaremos esta seção retomando alguns conceitos que foram vistos no início deste livro, mais especificamente na Seção 1 da Unidade 1. Nessa seção, foram apresentados os três tipos de memórias pertencentes ao microcontrolador ATmega328, sendo elas: memória RAM, FLASH e EEPROM.

A memória RAM é aquela que utilizamos para o armazenamento de dados voláteis da nossa aplicação, ou seja, aqueles dados que serão modificados com maior frequência, que também podem ser perdidos quando o microcontrolador é desenergizado, como é o caso das variáveis de programa. A grande vantagem dessas memórias é a sua velocidade de acesso para leitura e escrita, e sua principal desvantagem é a não persistência dos dados na falta de alimentação do sistema.

Já as memórias FLASH são aquelas com maior espaço disponível dentro do nosso microcontrolador e seus dados não são perdidos no desligamento do sistema. Ela é utilizada majoritariamente para o armazenamento das instruções de seu programa, podendo também ser utilizada para armazenarmos dados em nosso microcontrolador que não necessitarão ser sobrescritos, somente lidos, como tabelas.

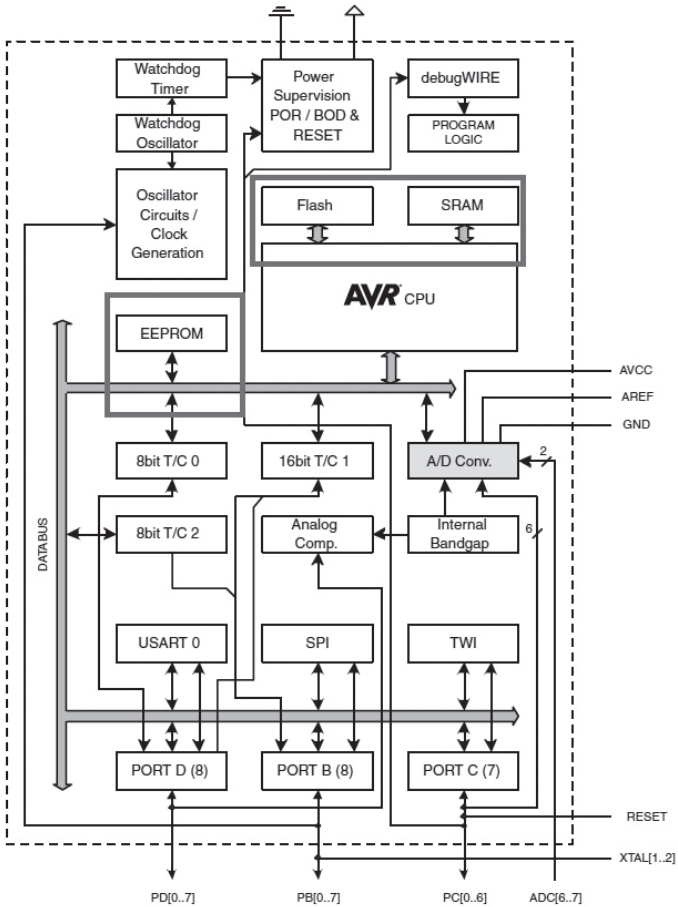
Assimile

As memórias FLASH são uma evolução das memórias EEPROM, sendo mais velozes e versáteis. Porém, sempre que se faz necessário sobrescrever o conteúdo de algum endereço dessas memórias, todo um bloco de endereços deve ser apagado primeiramente, sendo que tais blocos podem possuir tamanhos diversos, desde algumas centenas de endereços, até milhares.

Finalmente, as memórias EEPROM possuem características de não volatilidade, semelhantes à memória FLASH, tendo como vantagem o acesso individual de leitura e escrita para cada byte, não necessitando assim que todo um bloco de endereços seja apagado. Sua principal desvantagem é o seu tempo de acesso, sendo necessários até 3,4 ms para uma simples escrita. Esse tipo de memória é muito utilizado para o armazenamento de dados críticos para o sistema, e que não podem ser perdidos ou apagados quando ocorre o seu desligamento.

Estes três blocos de memória descritos estão conectados ao microcontrolador em barramentos distintos, como podemos ver no diagrama de blocos do microcontrolador (Figura 4.21).

Figura 4.21 | Disposição das memórias no Microcontrolador ATmega328



Fonte: Atmel AVR (2016, p. 6).

Pesquise mais

As memórias eletrônicas se tornaram um elemento vital no nosso cotidiano, sendo encontradas em praticamente qualquer produto que possua processamento digital. Porém, no início, o seu poder de armazenamento e velocidade de acesso eram muito limitados em comparação com a tecnologia atual, e uma pergunta pode ser feita: até qual patamar a tecnologia das memórias poderá evoluir no futuro? Veja

mais sobre a história das memórias nesse documentário: **Maravilhas Modernas: a evolução da memória.** Disponível em: <https://www.youtube.com/watch?v=vlY2fSwkUYQ>. Acesso em: 27 out. 2017.

Memória EEPROM no ATmega328

A memória EEPROM do microcontrolador ATmega328 possui um total de 1 Kbyte (1024 bytes) de endereços para armazenamento de dados, sendo que cada byte pode ser sobrescrito minimamente até 100.000 vezes.

O acesso à memória EEPROM é feito através da memória de I/O, ou seja, o mesmo espaço de memória utilizado para o acesso e a configuração dos demais periféricos do microcontrolador. São reservados, desta forma, registradores específicos para controlar a leitura e a escrita de dados, e também para direcionar em qual endereço o valor deve ser armazenado. Os endereços da memória EEPROM vão desde o valor 0x00 (0 em decimal) até 0x3FF (1023 em decimal).

Reflita

Apesar de possuir até 100 mil ciclos de escrita e leitura, esse número é finito, e quando ultrapassado pode gerar corrupção dos dados armazenados e lidos. Como podemos evitar que isso ocorra em projetos que utilizam a memória EEPROM frequentemente?

A memória EEPROM é um periférico cuja estabilidade de funcionamento depende muito do nível de tensão de alimentação. Em algumas condições, dados corrompidos podem ser escritos ou lidos dessa memória pelo fato de a tensão de entrada (Vcc) ser inferior ao mínimo adequado para operação do microcontrolador. Desta forma, um cuidado especial com as fontes de alimentação deve ser tomado em projetos em que se deseja utilizar a memória EEPROM para o armazenamento de dados críticos.

Processo de leitura e escrita da memória EEPROM

O processo de leitura e escrita da memória EEPROM é relativamente simples, sendo que somente alguns cuidados devem ser tomados para garantir a integridade dos dados lidos ou armazenados. Da mesma forma, é fundamental que, durante esse processo, nenhuma interrupção ocorra. Assim, o bit de interrupção global I do registrador SREG deve ser desabilitado momentaneamente, e habilitado logo após o término da operação.

De acordo com os registradores e bits de trabalho da EEPROM, os seguintes passos são necessários para efetuar uma escrita (a ordem dos passos 2 e 3 não é importante). Esses passos são necessários para se evitar uma eventual escrita acidental (LIMA; VILLAÇA, 2012).

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.

Exemplificando

Neste exemplo, é mostrado como realizar a leitura e a escrita da memória EEPROM em Linguagem C e Assembly.

Linguagem C:

```
#include <avr/interrupt.h>

void EEPROM_escrita(unsigned int Endereco, unsigned char Dado){
  cli(); //Desabilita a interrupção global
```

```
  while(EECR & (1<<EEPE)); //espera completar a escrita anterior
  EEAR = Endereco; //carrega o endereço para a escrita
  EEDR = Dado; //carrega o dado a ser escrito
  EECR |= (1<<EEMPE); //habilita o bit EEMPE
  EECR |= (1<<EEPE); //inicia a escrita habilitando o bit EEPE
  sei(); //Habilita a interrupção global
}
unsigned char EEPROM_leitura(unsigned int Endereco){
  cli(); //Desabilita a interrupção global
  while(EECR & (1<<EEPE)); //espera completar a escrita anterior
  EEAR = Endereco; //carrega o endereço de leitura
  EECR |= (1<<EERE); //inicia a leitura ativando o bit EERE
  sei(); //Habilita a interrupção global
  return EEDR; //retorna o valor lido do registrador de dados
}
```

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
```

Registradores do periférico EEPROM

EEARH e EEARL

Figura 4.22 | Registrador EEAR

–	–	–	–	–	–	EEAR9	EEAR8	EEARH
EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL

Fonte: elaborada pelo autor.

- EEAR9...0: Valor de 0x00 a 0x3FF que representa o endereço no qual se deseja acessar a memória EEPROM para escrita ou para leitura.

EECR

Figura 4.23 | Registrador EECR

–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	EECR

Fonte: elaborada pelo autor.

- EEPM1 e EEPM0: a configuração desses bits define qual tipo de operação deve ser realizada quando o bit EEPE for habilitado. EEPM = {0,0} → Apaga o valor presente no endereço e faz a escrita do novo valor em uma única operação; EEPM = {0,1} → Somente apaga o valor; EEPM = {1,0} → Somente faz a escrita do valor; EEPM = {1,1} → Não válido.
- EERIE: bit que habilita o modo de interrupção da EEPROM. A interrupção ocorre quando a EEPROM está pronta para uma nova operação.
- EEMPE: quando esse bit é ativo (lógica 1), habilita a escrita na memória EEPROM.
- EEPE: quando esse bit é ativo e o endereço e o valor a serem escritos na memória EEPROM são válidos, então o processo de escrita é iniciado.

Biblioteca eeprom.h

Para projetos escritos em linguagem C, o compilador utilizado pelo programa ATMEL Studio, o AVR-GCC disponibiliza a biblioteca eeprom.h, presente na pasta avr (se declara como #include <avr/eeprom.h>. Esta biblioteca implementa várias funções muito úteis para se poupar tempo de desenvolvimento, com a garantia de que o código seja robusto e ágil.

A biblioteca também implementa o atribuído EEMEM, que sinaliza ao compilador que as variáveis devem ser armazenadas na memória EEPROM. Da mesma forma, um arquivo *.eep é gerado com as diretivas para a gravação dessas variáveis.

Exemplificando

Neste exemplo a seguir, é mostrado como utilizar as funções disponíveis na biblioteca eeprom.h e também como utilizar o atributo EEMEM para declaração de variáveis que serão armazenadas nessa memória (LIMA; VILLAÇA, 2012):

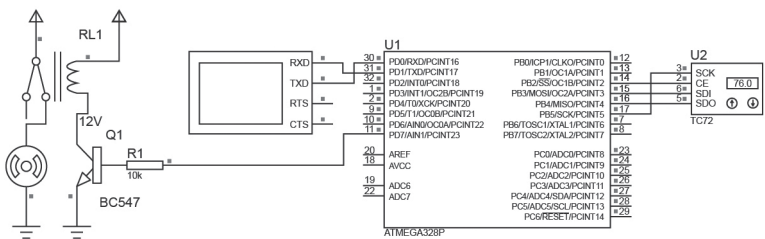
```
//============================================================================== //
#include <avr/io.h>
#include <avr/eeprom.h>
//------------------------------------------------------------------------------
/*inicialização dos valores para a EEPROM começando no endereço 0x00 (default), não
consome 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 ucDdado_leitura;
     unsigned int  uiEndereco = 0x3FF; /*endereço 0x3FF (1024° posição da memória)
                                                    valores entre 0 e 1023*/
     _EEPUT(uiEndereco,ucDado_escrita);  //gravando 0x13 no endereço 0x3FF da EEPROM
     _EEGET(ucDdado_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
}
//==============================================================================
```

Sem medo de errar

Neste seu último e mais avançado projeto em sistemas embarcados, lhe foi solicitado que o projeto de controle de temperatura fosse revisado, adicionando uma memória EEPROM para o armazenamento de dados importantes do projeto. Dessa forma, pede-se para utilizar o sensor de temperatura TC72 para realizar leituras e armazenar os seguintes valores em memória RAM (a cada 100 ms) e memória EEPROM (a cada 10 s): a) a última leitura; b) o menor valor e c) o maior valor. Pelo terminal serial, temos os seguintes comandos de tecla: a) 's': retorna o valor de temperatura atual, máxima, mínima e temperatura configurada; b) 'v': retorna o estado da ventoinha; c) 'u': aumenta o valor da temperatura configurada em 5 °C até o máximo de 120 °C, e salva em EEPROM; d) 'd': diminui o valor da temperatura configurada em 5 °C até o mínimo de 25 °C, e também salva em EEPROM e e) 'z': para zerar todos os valores da memória EEPROM. Nesse novo cenário, sempre que o sistema for reiniciado, o microcontrolador deve buscar da memória EEPROM os valores de temperatura mínima, máxima e configurada.

Iniciaremos a resolução do projeto utilizando o mesmo diagrama de hardware da última versão do projeto.

Figura 4.24 | Diagrama de hardware



Fonte: elaborada pelo autor.

Como um primeiro passo para a solução de software, dividiremos o nosso projeto em três arquivos distintos, de forma a organizar cada qual segundo um escopo, sendo: main.c para a lógica principal, funcoes.c para as funções de comunicação, e defs.h para a declaração de macros e protótipos de função. Em

nosso código-fonte principal, consideraremos o armazenamento de quatro variáveis na memória EEPROM, conforme solicitado pelos requisitos, sendo três delas para as temperaturas estatísticas e de configuração, e uma dessas será utilizada para definir se a memória EEPROM já foi escrita antes ou não. Essa informação é importante, pois quando o sistema reiniciar, teremos que atualizar nossas variáveis RAM com os valores que foram salvos anteriormente na EEPROM, e caso seja a primeira vez que esse procedimento é realizado, as variáveis RAM receberão valores aleatórios como resposta. Adicionaremos também um novo estado de leitura para a tecla 'z', responsável por zerar os valores presentes na memória EEPROM, e a cada 10 segundos salvaremos os valores solicitados de temperatura da memória RAM para a memória EEPROM. A seguir, uma possível solução para o problema:

Figura 4.25 | Arquivo main.c

```
#include "defs.h"

unsigned char dado_recebido, dado_envia, contadorEEP;
unsigned char tempInt, tempDec, MaxTemp, MinTemp;
unsigned char TempControl = 50;
unsigned char comando, vent, PrimeiraEscrita;
//Declaração de um vetor de caracteres - string
char string[60] = {0x00};

//Declaração das variáveis que serão salvas em EEPROM
unsigned char EEMEM TempMinEEP = 0x00;
unsigned char EEMEM TempMaxEEP = 0x01;
unsigned char EEMEM TempControlEEP = 0x02;
unsigned char EEMEM PrimeiraEscritaEEP = 0x03;

int main(void)
{
  int i = 0;
  //Inicia USART
  USART_config(MYUBRR);
  //Inicia SPI
  SPI_Mestre_config();
  set_bit(DDRD, PD7);
  //Configura sensor de temperatura
  TC72_ON();
  SPI(0x80);
  SPI(0x00);
  TC72_OFF();
  /*Chega se é o sistema já foi ligado antes para
  ler os valores de EEPROM com segurança*/
  PrimeiraEscrita =
eeprom_read_byte(&PrimeiraEscritaEEP);
  if (PrimeiraEscrita == 0x01)
  {
    MinTemp = eeprom_read_byte(&TempMinEEP);
    MaxTemp = eeprom_read_byte(&TempMaxEEP);
    TempControl = eeprom_read_byte(&TempControlEEP);
  }
    while (1)
    {
    //Faz a leitura do sensor
    TC72_ON();
    SPI(0x02);
    tempInt = SPI(0x00);
    tempDec = SPI(0x00);
    TC72_OFF();
    /*Compara o valor de temperatura lido com os
valores de máximo e mínimo*/
    if(tempInt < MinTemp) {MinTemp = tempInt;}
    if(tempInt > MaxTemp) {MaxTemp = tempInt;}
    if(tempInt > TempControl) {VENT_ON(); vent = 1;}
    else {VENT_OFF(); vent = 0;}
    //Le se há um novo caracter na USART
    comando = USART_le();
    switch(comando)
    {
      //Se for 's' apresenta os valores já armazenados
      case 's':
        sprintf(string, "T. Atual: %dC T.Max: %dC
T.Min: %dC T.Set: %dC
\r\n",tempInt,MaxTemp,MinTemp,TempControl);
        while(string[i] != 0x00)
{USART_escreve(string[i]); i++;}
      break;
      //Se for 'v' indica os status da ventoinha
      case 'v':
        vent == 1 ? sprintf(string, "Ventoinha ligada!
\r\n") : sprintf(string, "Ventoinha desligada! \r\n");
        while(string[i] != 0x00)
{USART_escreve(string[i]); i++;}
      break;
      //Se for 'u' incrementa a temperatura em 5ºC
      case 'u':
        if(TempControl == 120) {TempControl = 120;}
else {TempControl += 5;}
      break;
      //Se for 'd' decrementa a temperatura em 5ºC
      case 'd':
        if(TempControl == 25) {TempControl = 25;} else
{TempControl -= 5;}
      break;
      case 'z':
      //Se for 'z' limpa os valores da EEPROM
        eeprom_write_byte(&TempMinEEP,0x00);
        eeprom_write_byte(&TempMaxEEP,0x00);
        eeprom_write_byte(&TempControlEEP,0x00);
        eeprom_write_byte(&PrimeiraEscritaEEP,0x01);
      break;
    }
    string[0] = 0x00;
    i = 0;
    _delay_ms(100);
    //Incrementa a variável a cada 100ms segundos
    contadorEEP += 1;
    //Se a variável chegar a 100 (10s), salva os
valores atuais em EEPROM
    if (contadorEEP >= 100)
    {
      contadorEEP = 0;
      eeprom_write_byte(&TempMinEEP,MinTemp);
      eeprom_write_byte(&TempMaxEEP,MaxTemp);
      eeprom_write_byte(&TempControlEEP,TempControl);
      eeprom_write_byte(&PrimeiraEscritaEEP,0x01);
    }
    }
}
```

Fonte: elaborada pelo autor.

Figura 4.26 | Arquivos funcoes.c e defs.c, respectivamente

```
#include "defs.h"

void SPI_Mestre_config(){
  DDRB = (1<<PB5)|(1<<PB3)|(1<<PB2);
  SPCR = (1<<SPE)|(1<<MSTR)|(SPR1)|(1<<CPHA);
}
unsigned char SPI(unsigned char pacote){
  SPDR = pacote;
  while(!(SPSR & (1<<SPIF)));
  return SPDR;
}

void USART_config(unsigned int usart)
{
  UBRR0H = (unsigned char)(MYUBRR>>8);
  UBRR0L = (unsigned char)MYUBRR;
  UCSR0B = (1<<RXEN0)|(1<<TXEN0);
  UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
}
unsigned char USART_le(void)
{
  if(UCSR0A & (1<<RXC0)) {return UDR0;} else {return
0x00;}
}
void USART_escreve(unsigned char caracter)
{
  while (!( UCSR0A & (1<<UDRE0)));
  UDR0 = caracter;
}
```

```
#ifndef DEFS_H_
#define DEFS_H_

#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include <stdio.h>
#include <avr/eeprom.h>

#define set_bit(Reg,bit) (Reg|=(1<<bit))
#define reset_bit(Reg,bit) (Reg&=~(1<<bit))
#define troca_bit(Reg,bit) (Reg^=(1<<bit))
#define le_bit(Reg,bit) ((Reg>>bit)&0x01)

#define VENT_ON()    set_bit(PORTD, PD7)
#define VENT_OFF()   reset_bit(PORTD, PD7)
#define TC72_ON()    set_bit(PORTB,PB2)
#define TC72_OFF()   reset_bit(PORTB,PB2)

#define TAXA_TRANS  19200
#define MYUBRR (F_CPU/16/TAXA_TRANS-1)

unsigned char USART_le(void);
void SPI_Mestre_config();
unsigned char SPI(unsigned char pacote);
unsigned char USART_le(void);
void USART_escreve(unsigned char caracter);
void USART_escreve(unsigned char caracter);
void USART_config(unsigned int usart);

#endif /* DEFS_H_ */
```

Fonte: elaborada pelo autor.

Avançando na prática

Armazenamento de Strings em EEPROM

Descrição da situação-problema

O cliente do seu último projeto ficou muito satisfeito com o seu desenvolvimento, e decidiu solicitar mais uma melhoria pontual. Ele gostaria, ao iniciar o sistema, que fosse apresentado no terminal serial uma mensagem de saudação ao usuário, no estilo: "BEM-VINDO NOME_DO_USUARIO!". Como ele sabe que os produtos sairão da linha de produção sem a gravação predefinida do nome do usuário, lhe foi solicitado que, ao identificar que é a primeira vez que o sistema é energizado, o computador ao usuário que digite seu nome, e após, pressione a tecla ENTER para realizar a gravação. Qual seria uma possível solução de software para esse projeto?

Resolução da situação-problema

Utilizando o mesmo diagrama da Figura 4.24, e os mesmos arquivos apresentados na Figura 4.26, segue uma possível solução de software para o problema.

Figura 4.27 | Arquivo main.c para a solução do problema

```
#include <avr/io.h>
#include "defs.h"

unsigned char EEMEM PriEscritaEEP = {0x00};
unsigned char PriEscritaRAM = {0x00};
char EEMEM NomeEEP[60] = {0x00};
char NomeRAM[60] = {0x00};
char Saudacao[60] = {0x00};
unsigned char QuantLetras = 0;
unsigned char EEMEM QuantLetrasEEP = 0;
char letra = 0xFF;

int main(void)
{
  int i = 0;
  USART_config(MYUBRR);
  PriEscritaRAM = eeprom_read_byte(&PriEscritaEEP);
  while (1)
  {
    //Checa se o nome ainda não foi gravado em EEPROM
    if (PriEscritaRAM != 0x01)
    {
      //Exibe imagem de saudação
      sprintf(Saudacao, "Por favor, digite seu nome e pressione a tecla ENTER! \r\n");
      while(Saudacao[i] != 0x00) {USART_escreve(Saudacao[i]); i++;}
      //Laço para ler todas as teclas digitadas, até que o usuário pressione ENTER (0x0D)
      do
      {
        letra = USART_le();
        if(letra != 0x00)
        {
          USART_escreve(letra);
          NomeRAM[QuantLetras++] = letra;
        }
      } while (letra != 0x0D);


      //Salva o nome em EEPROM
      eeprom_write_block(NomeRAM,NomeEEP,QuantLetras);
      eeprom_write_byte(&PriEscritaEEP,0x01);
      eeprom_write_byte(&QuantLetrasEEP,QuantLetras);
      eeprom_write_block()

      sprintf(Saudacao, "Obrigado!\r\n");
      while(Saudacao[i] != 0x00) {USART_escreve(Saudacao[i]); i++;}
    }
    else
    {
      //Se o nome já foi salvo, exibe a saudação com o nome
      QuantLetras = eeprom_read_byte(&QuantLetrasEEP);
      eeprom_read_block(NomeRAM,NomeEEP,QuantLetras);
      sprintf(Saudacao, "BEM-VINDO %s! \r\n", NomeRAM);
      while(Saudacao[i] != 0x00) {USART_escreve(Saudacao[i]); i++;}
    }
  }
}
```

Fonte: elaborada pelo autor.

Faça valer a pena

1. As memórias eletrônicas desempenham um papel fundamental no funcionamento do microcontrolador, sendo que cada tecnologia é utilizada para atender a um aspecto do projeto embarcado.
Complete as lacunas da afirmação seguinte:

A memória _____ é aquela utilizada para armazenar dados voláteis, como variáveis e estado do software, diferentemente da memória _____, que também pode ser utilizada para armazenar variáveis e dados, só que de maneira permanente. Um outro tipo de memória fundamental ao microcontrolador é a memória _____, que é utilizada para armazenar o programa de execução.

a) EEPROM, FLASH, RAM.
b) RAM, EEPROM, FLASH.
c) FLASH, RAM, EEPROM.
d) EEPROM, RAM, FLASH.
e) FLASH, EEPROM, RAM.

2. A memória EEPROM, mesmo sendo antecessora em termos tecnológicos à memória FLASH, ainda é muito utilizada em projetos para o armazenamento de dados cruciais para a correta operação do sistema, mesmo em condições adversas de alimentação do sistema.
Em um projeto de software embarcado para aquisição de sinais de um sensor analógico, o engenheiro responsável decide criar três estruturas de dados para auxiliá-lo na operação do sistema: uma variável para armazenar o valor atual de leitura, que é feito a cada 10 ms, outra variável para armazenar o máximo valor lido pelo sistema, e também uma tabela contendo a relação de senos e cossenos para vários valores de ângulos que serão utilizados no cálculo de controle do sistema.
Com base nessas informações, é recomendável que o engenheiro utilize quais tipos de memória para o armazenamento de cada estrutura de dados?

a) O valor atual deveria ser armazenado na memória EEPROM, o valor máximo na RAM e a tabela na memória FLASH.
b) A tabela poderia ser armazenada na memória FLASH, e o valor atual e de máximo na memória RAM.
c) Todos os valores deveriam ser armazenados exclusivamente na memória RAM, dada a sua velocidade de acesso.
d) Não haveria restrições em armazenar os valores na memória EEPROM, dada a vantagem de que os valores nunca seriam perdidos, mesmo com a falta de alimentação do sistema.
e) O valor atual deveria ser armazenado na memória RAM, o valor de máximo poderia ser armazenado na memória EEPROM, e a tabela poderia ser armazenada na memória FLASH.

3. O microcontrolador ATmega328 possui uma memória EEPROM com quantidade de endereços expressiva, que pode ser acessada através do mapeamento de I/O presente nos seus registradores, facilitando, dessa forma, a interface entre o software aplicativo e o hardware em si.
Sobre a memória EEPROM do microcontrolador ATmega328, qual das afirmações a seguir é falsa?

a) A memória EEPROM possui ao todo 1024 endereços disponíveis para escrita e leitura de dados.
b) É possível carregar valores na memória EEPROM pelo software simplesmente incluindo a biblioteca eeprom.h e o atributo EEMEM nas variáveis de interesse.
c) A única diferença entre a memória EEPROM do microcontrolador ATmega328 e sua memória RAM é a volatilidade dos dados quanto à perda de alimentação.
d) É possível utilizar várias funções de manipulação da memória EEPROM, adicionando a biblioteca eeprom.h.
e) É possível configurar uma interrupção a ser acionada sempre que a memória EEPROM estiver pronta para um novo processo de escrita.

Referências

ATMEL AVR. **ATmega328 Datasheet Complete. Atmel-42735B.** 2016.

LIMA, C. B. D.; VILLAÇA, M. V. M. **AVR e Arduino técnicas de projeto**. Florianópolis: Publicação independente, 2012.

SOARES, M. **Uso dos registros UCSRA, UCSRB, UCSRC, UDR, UBRRL e UBRRH na comunicação assíncrona.** [s.d.]. Disponível em: <http://www.arnerobotics.com.br/eletronica/Configurando_corretamente_regsAVR_pt5.htm>. Acesso em: 27 out. 2017.

Anotações

Anotações

Anotações

Anotações

ISBN 978-85-522-0273-8